



IBM Espresso RISC Microprocessor Developer's User Manual

Version 0.2

Advance

September 28, 2011—IBM Confidential—Available under NDA only



© Copyright International Business Machines Corporation 2011

Printed in the United States of America September 2011

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Other company, product, and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

Note: This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice. Verify with your IBM field applications engineer that you have the latest version of this document before finalizing a design.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at ibm.com®.
The IBM semiconductor solutions home page can be found at ibm.com/chips.

Version 0.2
September 28, 2011—IBM Confidential—Available under NDA only

Contents

List of Figures	9
List of Tables	11
Revision Log	15
Preface	17
1. Espresso Overview	19
1.1 Espresso Microprocessor Overview	19
1.1.1 On-Chip L2 Cache Implementation	20
1.1.2 Core Interface Unit	21
1.2 Espresso Core Features	22
1.2.1 Overview of Espresso Core Features	24
1.2.2 Instruction Flow	27
1.2.2.1 Instruction Queue and Dispatch Unit	27
1.2.2.2 Branch Processing Unit	27
1.2.2.3 Completion Unit	28
1.2.2.4 Independent Execution Units	29
1.2.3 Memory Management Units (MMUs)	30
1.2.4 On-Chip L1 Instruction and Data Caches	31
1.3 Espresso Microprocessor Implementation	33
1.4 PowerPC Registers and Programming Model	35
1.5 Instruction Set	38
1.5.1 PowerPC Instruction Set	38
1.5.2 Espresso Microprocessor Instruction Set	39
1.6 On-Chip Cache Implementation	40
1.6.1 PowerPC Cache Model	40
1.6.2 Espresso Microprocessor Cache Implementation	40
1.7 Exception Model	41
1.7.1 PowerPC Exception Model	41
1.7.2 Espresso Microprocessor Exception Implementation	42
1.8 Memory Management	43
1.8.1 PowerPC Memory Management Model	43
1.8.2 Espresso Memory Management Implementation	44
1.9 Instruction Timing	44
1.10 Performance Monitor	46
2. Programming Model	49
2.1 Espresso Processor Register Set	49
2.1.1 Register Set	49
2.1.2 Espresso-Specific Registers	55
2.1.2.1 Instruction Address Breakpoint Register (IABR)	55
2.1.2.2 Processor ID Register (PIR)	56
2.1.2.3 Performance Monitor Registers	57

2.1.2.4 Direct Memory Access Registers (DMAU and DMAL)	62
2.1.2.5 Graphics Quantization Registers (GQRs)	64
2.2 Operand Conventions	65
2.2.1 Data Organization in Memory and Data Transfers	65
2.2.2 Alignment and Misaligned Accesses	65
2.2.3 Floating-Point Operand and Execution Models—UISA	66
2.3 Instruction Set Summary	71
2.3.1 Classes of Instructions	72
2.3.1.1 Definition of Boundedly Undefined	72
2.3.1.2 Defined Instruction Class	72
2.3.1.3 Illegal Instruction Class	73
2.3.1.4 Reserved Instruction Class	73
2.3.1.5 Espresso Implementation-Specific Instructions	74
2.3.2 Addressing Modes	74
2.3.2.1 Memory Addressing	74
2.3.2.2 Memory Operands	74
2.3.2.3 Effective Address Calculation	75
2.3.2.4 Synchronization	75
2.3.3 Instruction Set Overview	76
2.3.4 PowerPC UISA Instructions	77
2.3.4.1 Integer Instructions	77
2.3.4.2 Floating-Point Instructions	80
2.3.4.3 Load and Store Instructions	84
2.3.4.4 Branch and Flow Control Instructions	96
2.3.4.5 System Linkage Instruction (UISA)	98
2.3.4.6 Processor Control Instructions (UISA)	98
2.3.4.7 Memory Synchronization Instructions (UISA)	103
2.3.5 PowerPC VEA Instructions	104
2.3.6 Processor Control Instructions (VEA)	104
2.3.6.1 Memory Synchronization Instructions (VEA)	104
2.3.6.2 Memory Control Instructions (VEA)	105
2.3.6.3 Optional External Control Instructions	108
2.3.7 PowerPC OEA Instructions	108
2.3.7.1 System Linkage Instructions (OEA)	108
2.3.7.2 Processor Control Instructions (OEA)	109
2.3.7.3 Memory Control Instructions (OEA)	109
2.3.8 Recommended Simplified Mnemonics	110
3. Espresso Instruction and Data Cache Operation	111
3.1 Data Cache Organization	113
3.2 Instruction Cache Organization	114
3.3 Memory and Cache Coherency	116
3.3.1 Memory/Cache Access Attributes (WIMG Bits)	116
3.3.2 MESI Protocol	117
3.3.2.1 MESI Hardware Considerations	119
3.3.3 Coherency Precautions	120
3.3.4 Espresso-Initiated Load/Store Operations	120
3.3.4.1 Performed Loads and Stores	120
3.3.4.2 Sequential Consistency of Memory Accesses	121
3.3.4.3 Atomic Memory References	121

3.4 Cache Control	122
3.4.1 Cache Control Instructions	122
3.4.1.1 Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst)	122
3.4.1.2 Data Cache Block Zero (dcbz)	123
3.4.1.3 Data Cache Block Store (dcbst)	123
3.4.1.4 Data Cache Block Flush (dcbf)	123
3.4.1.5 Data Cache Block Invalidate (dcbi)	124
3.4.1.6 Instruction Cache Block Invalidate (icbi)	124
3.5 Cache Operations	124
3.5.1 Cache Block Replacement/Flush Operations	124
3.5.2 Cache Flush Operations	127
3.5.3 Data Cache-Block-Fill Operations	127
3.5.4 Instruction Cache-Block-Fill Operations	128
3.5.5 Data Cache-Block-Push Operation	128
3.6 L1 Caches and Bus Transactions	128
3.6.1 Snooping	128
3.7 MESI State Transactions	130
4. Exceptions	133
4.1 Microprocessor Exceptions	134
4.2 Exception Recognition and Priorities	135
4.3 Exception Processing	138
4.3.1 Machine Status Save/Restore Register 0 (SRR0)	138
4.3.2 Machine Status Save/Restore Register 1 (SRR1)	138
4.3.3 Machine State Register (MSR)	139
4.3.4 Enabling and Disabling Exceptions	141
4.3.5 Steps for Exception Processing	141
4.3.6 Setting MSR[RI]	142
4.3.7 Returning from an Exception Handler	142
4.4 Process Switching	143
4.5 Exception Definitions	143
4.5.1 Machine Check Exception (x'00200')	144
4.5.1.1 Machine Check Exception Enabled (MSR[ME] = '1')	145
4.5.1.2 Checkstop State (MSR[ME] = '0')	145
4.5.2 DSI Exception (x'00300')	145
4.5.3 ISI Exception (x'00400')	146
4.5.4 External Interrupt Exception (x'00500')	146
4.5.5 Alignment Exception (x'00600')	147
4.5.6 Program Exception (x'00700')	147
4.5.7 Floating-Point Unavailable Exception (x'00800')	148
4.5.8 Decrementer Exception (x'00900')	148
4.5.9 System Call Exception (x'00C00')	148
4.5.10 Trace Exception (x'00D00')	148
4.5.11 Floating-Point Assist Exception (x'00E00')	149
4.5.12 Performance Monitor Interrupt (x'00F00')	149
4.5.13 Instruction Address Breakpoint Exception (x'01300')	150

5. Memory Management	151
5.1 MMU Overview	152
5.1.1 Memory Addressing	153
5.1.2 MMU Organization	154
5.1.3 Address Translation Mechanisms	158
5.1.4 Large Page Support	160
5.1.4.1 Page Table Lookup Procedure	160
5.1.5 Memory Protection Facilities	161
5.1.6 Page History Information	162
5.1.7 General Flow of MMU Address Translation	162
5.1.7.1 Real Addressing Mode and Block Address Translation Selection	162
5.1.7.2 Page Address Translation Selection	164
5.1.8 MMU Exceptions Summary	166
5.1.9 MMU Instructions and Register Summary	168
5.2 Real Addressing Mode	170
5.3 Block Address Translation	171
5.4 Memory Segment Model	171
5.4.1 Page History Recording	172
5.4.1.1 Referenced Bit	172
5.4.1.2 Changed Bit	173
5.4.1.3 Scenarios for Referenced and Changed Bit Recording	173
5.4.2 Page Memory Protection	174
5.4.3 TLB Description	174
5.4.3.1 TLB Organization	175
5.4.3.2 TLB Invalidation	176
5.4.4 Page Address Translation Summary	177
5.4.5 Page Table Search Operation	179
5.4.6 Page Table Updates	182
5.4.7 Segment Register Updates	182
6. Instruction Timing	183
6.1 Terminology and Conventions	183
6.2 Instruction Timing Overview	185
6.3 Timing Considerations	188
6.3.1 General Instruction Flow	189
6.3.2 Instruction Fetch Timing	190
6.3.2.1 Cache Arbitration	190
6.3.2.2 Cache Hit	190
6.3.2.3 Cache Miss	195
6.3.2.4 L2 Cache Access Timing Considerations	197
6.3.2.5 Instruction Dispatch and Completion Considerations	197
6.3.2.6 Rename Register Operation	198
6.3.2.7 Instruction Serialization	198
6.4 Execution Unit Timings	199
6.4.1 Branch Processing Unit Execution Timing	199
6.4.1.1 Branch Folding	199
6.4.1.2 Branch Instructions and Completion	201
6.4.1.3 Branch Prediction and Resolution	201
6.4.2 Integer Unit Execution Timing	205

6.4.3 Floating-Point Unit Execution Timing	205
6.4.4 Effect of Floating-Point Exceptions on Performance	205
6.4.5 Load/Store Unit Execution Timing	206
6.4.6 Effect of Operand Placement on Performance	206
6.4.7 Integer Store Gathering	207
6.4.8 System Register Unit Execution Timing	207
6.5 Memory Performance Considerations	208
6.5.1 Caching and Memory Coherency	208
6.5.2 Effect of TLB Miss	209
6.6 Instruction Scheduling Guidelines	209
6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements	210
6.6.1.1 Branch Resolution Resource Requirements	210
6.6.1.2 Dispatch Unit Resource Requirements	210
6.6.1.3 Completion Unit Resource Requirements	210
6.7 Instruction Latency Summary	211
7. Memory Subsystem Operation	219
7.1 Memory Interface Unit	219
7.1.1 Operation of the Instruction and Data L1 Caches	221
7.2 Core Interface Unit	222
7.2.1 Intervention	223
7.2.1.1 Cache Coherency States	223
7.2.1.2 Snoop Handling	224
7.2.1.3 Intervention Responses	226
7.3 Bus Interface Unit Overview	228
8. L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe	229
8.1 L2 Cache Overview	229
8.1.1 L2 Cache Tag Array	230
8.1.2 Cache Coherency	232
8.1.2.1 Tag Updates	234
8.1.3 Dataflow	234
8.1.4 L2 Cache Operation	236
8.1.4.1 32-Byte Fetch Mode	236
8.1.4.2 64-Byte Fetch Mode	238
8.2 Locked L1 Data Cache	238
8.2.1 Locked Cache Configuration	238
8.2.2 Locked Cache Operation	239
8.2.2.1 dcbz	239
8.2.2.2 dcbz_l	239
8.2.2.3 dcbi	240
8.2.2.4 dcbf	240
8.2.2.5 dcbst	240
8.2.2.6 dcbt and dcbtst	240
8.2.2.7 Load and Store	240
8.3 Direct Memory Access (DMA)	240
8.3.1 DMA Operation	241
8.3.2 Exception Conditions	241
8.3.2.1 DMA Queue Overflow	242

8.3.2.2 DMA Look-up Hits Normal Cache	242
8.3.2.3 DMA Look-up Miss	242
8.3.3 DMA Timing	242
9. Performance Monitor	243
9.1 Performance Monitor Interrupt	244
9.2 Special-Purpose Registers Used by Performance Monitor	244
9.2.1 Monitor Mode Control Register 0 (MMCR0)	245
9.2.2 User Monitor Mode Control Register 0 (UMMCR0)	246
9.2.3 Monitor Mode Control Register 1 (MMCR1)	247
9.2.4 User Monitor Mode Control Register 1 (UMMCR1)	247
9.2.5 Performance Monitor Counter Registers (PMC1 - PMC4)	247
9.2.6 User Performance Monitor Counter Registers (UPMC1 - UPMC4)	252
9.2.7 Sampled Instruction Address Register (SIA)	252
9.2.8 User Sampled Instruction Address Register (USIA)	252
9.3 Event Counting	253
9.4 Event Selection	253
9.5 Notes	254
10. Espresso PowerPC Instruction Set	255
10.1 Instruction Formats	255
10.1.1 Split-Field Notation	255
10.1.2 Instruction Fields	256
10.1.3 Notation and Conventions	257
10.1.4 Computation Modes	260
10.2 PowerPC Instruction Set	261
Appendix A. Espresso Instruction Set	495
A.1 Instructions Sorted by Opcode	495
A.2 Instructions Grouped by Functional Categories	502
Glossary	513

List of Figures

Figure 1-1.	Espresso Microprocessor Block Diagram	20
Figure 1-2.	Espresso Core Block Diagram	23
Figure 1-3.	Cache Organization	32
Figure 1-4.	Pipeline Diagram	45
Figure 2-1.	Programming Model (User Model–VEA and UISA)—Espresso Microprocessor Registers	50
Figure 2-2.	Floating-Point Register Containing a Paired Single Operand	67
Figure 3-1.	Cache Integration	112
Figure 3-2.	Data Cache Organization	114
Figure 3-3.	Instruction Cache Organization	115
Figure 3-4.	MESI Cache Coherency Protocol State Diagram (WIM = '001')	118
Figure 3-5.	PLRU Replacement Algorithm	125
Figure 5-1.	MMU Conceptual Block Diagram	155
Figure 5-2.	Espresso Microprocessor IMMU Block Diagram	156
Figure 5-3.	Espresso Microprocessor DMMU Block Diagram	157
Figure 5-4.	Address Translation Types	159
Figure 5-5.	General Flow of Address Translation (Real Addressing Mode and Block)	163
Figure 5-6.	General Flow of Page and Direct-Store Interface Address Translation	165
Figure 5-7.	Segment Register and DTLB Organization	175
Figure 5-8.	Page Address Translation Flow (TLB Hit)	178
Figure 5-9.	Primary Page Table Search	180
Figure 5-10.	Secondary Page Table Search Flow	181
Figure 6-1.	Pipelined Execution Unit	185
Figure 6-2.	Superscalar/Pipeline Diagram	186
Figure 6-3.	Espresso Microprocessor Pipeline Stages	188
Figure 6-4.	Instruction Flow Diagram	191
Figure 6-5.	Instruction Timing - Cache Hit	193
Figure 6-6.	Instruction Timing - Cache Miss	196
Figure 6-7.	Branch Taken	200
Figure 6-8.	Removal of Fall-Through Branch Instruction	200
Figure 6-9.	Branch Completion	201
Figure 6-10.	Branch Instruction Timing	204
Figure 7-1.	Memory Interface Address Buffers	220
Figure 7-2.	CIU Block Diagram	222
Figure 8-1.	L2 Core Dataflow	235
Figure 10-1.	Instruction Description	261



List of Tables

Table 1-1.	Architecture-Defined Registers (Excluding SPRs)	35
Table 1-2.	Architecture-Defined SPRs Implemented	36
Table 1-3.	Implementation-Specific Registers	37
Table 1-4.	Implementation-Specific Shared Registers	37
Table 1-5.	Espresso Microprocessor Exception Classifications	42
Table 1-6.	Exceptions and Conditions	42
Table 2-1.	Additional MSR Bits	52
Table 2-2.	Additional SRR1 Bits	54
Table 2-3.	Quantized Data Types	64
Table 2-4.	Memory Operands	65
Table 2-5.	Floating-Point Operand Data Type Behavior	69
Table 2-6.	Floating-Point Unit Result Data Type Behavior	70
Table 2-7.	Integer Arithmetic Instructions	77
Table 2-8.	Integer Compare Instructions	78
Table 2-9.	Integer Logical Instructions	79
Table 2-10.	Integer Rotate Instructions	80
Table 2-11.	Integer Shift Instructions	80
Table 2-12.	Floating-Point Arithmetic Instructions	81
Table 2-13.	Floating-Point Multiply-Add Instructions	82
Table 2-14.	Floating-Point Rounding and Conversion Instructions	82
Table 2-15.	Floating-Point Comparison Instructions	83
Table 2-16.	Floating-Point Status and Control Register Instructions	83
Table 2-17.	Floating-Point Move Instructions	84
Table 2-18.	Integer Load Instructions	86
Table 2-19.	Integer Store Instructions	87
Table 2-20.	Integer Load and Store with Byte-Reverse Instructions	88
Table 2-21.	Integer Load/Store Multiple Instructions	88
Table 2-22.	Integer Load/Store String Instructions	89
Table 2-23.	Floating-Point Load Instructions	90
Table 2-24.	Floating-Point Store Instructions	91
Table 2-25.	Store Floating-Point Single Behavior	91
Table 2-26.	Store Floating-Point Double Behavior	92
Table 2-27.	Paired Single Load and Store Instructions	93
Table 2-28.	Conversion of Integer Value 1 to Single-Precision Floating-Point Value	94
Table 2-29.	Conversion of Floating-Point Value 1.00 E+2 to Integer	95
Table 2-30.	Branch Instructions	96
Table 2-31.	Condition Register Logical Instructions	97
Table 2-32.	Trap Instructions	97

Table 2-33.	System Linkage Instruction (UISA)	98
Table 2-34.	Move to/from Condition Register Instructions	98
Table 2-35.	Move to/from Special-Purpose Register Instructions (UISA)	98
Table 2-36.	SPR Encodings for PowerPC-Defined Registers	99
Table 2-37.	SPR Encodings for Espresso-Defined Registers	101
Table 2-38.	Memory Synchronization Instructions (UISA)	103
Table 2-39.	Move from Time Base Instruction	104
Table 2-40.	Memory Synchronization Instructions (VEA)	105
Table 2-41.	User-Level Cache Instructions	106
Table 2-42.	External Control Instructions	108
Table 2-43.	System Linkage Instructions (OEA)	108
Table 2-44.	Move to/from Machine State Register Instructions	109
Table 2-45.	Move to/from Special-Purpose Register Instructions (OEA)	109
Table 2-46.	Supervisor-Level Cache Management Instruction	110
Table 3-1.	MESI State Definitions	117
Table 3-2.	PLRU Bit Update Rules	126
Table 3-3.	PLRU Replacement Block Selection	126
Table 3-4.	Partitioned L1 PLRU Replacement Algorithm	127
Table 3-5.	L1 Data Cache State Transitions in Response to LSU Operations	130
Table 4-1.	Espresso Microprocessor Exception Classifications	134
Table 4-2.	Exceptions and Conditions	134
Table 4-3.	Espresso Exception Priorities	137
Table 4-4.	IEEE Floating-Point Exception Mode Bits	140
Table 4-5.	MSR Setting Due to Exception	143
Table 4-6.	HID0 Machine Check Enable Bits	144
Table 4-7.	Machine Check Exception (Register Settings)	145
Table 4-8.	Performance Monitor Interrupt Exception (Register Settings)	149
Table 4-9.	Instruction Address Breakpoint Exception (Register Settings)	150
Table 5-1.	MMU Feature Summary	152
Table 5-2.	Page Index and Offset Bits for Small and Large Pages	160
Table 5-3.	Access Protection Options for Pages	161
Table 5-4.	Translation Exception Conditions	166
Table 5-5.	Other MMU Exception Conditions for the Espresso Processor	167
Table 5-6.	Espresso Microprocessor Instruction Summary (Control MMUs)	168
Table 5-7.	Espresso Microprocessor MMU Registers	170
Table 5-8.	Table Search Operations to Update History Bits (TLB Hit Case)	172
Table 5-9.	Model for Guaranteed R and C Bit Settings	174
Table 6-1.	Performance Effects of Memory Operand Placement	206
Table 6-2.	TLB Miss Latencies	209

Table 6-3.	Branch Instructions	211
Table 6-4.	System Register Instructions	211
Table 6-5.	Condition Register Logical Instructions	212
Table 6-6.	Integer Instructions	213
Table 6-7.	Floating-Point Instructions	214
Table 6-8.	Load and Store Instructions	216
Table 7-1.	Cache Coherency States	223
Table 7-2.	CIU Snoop Handling of Memory Requests	224
Table 7-3.	Intervention Response to a Request from Core A (Normal Coherency Cases)	226
Table 7-4.	Intervention Response to a Request from Core A (Paradoxical Coherency Cases)	226
Table 7-5.	Instructions that Generate Intervention Requests	227
Table 7-6.	Coherency State Changes for Intervention Operations	227
Table 8-1.	Supported Cache Configurations	229
Table 8-2.	Address Bit Assignments for Different Cache Configurations	229
Table 8-3.	L2 Tag Array Entry Layout	230
Table 8-4.	L2 Tag Array Entry Bit Description	230
Table 8-5.	L2 PLRU Replacement Algorithm	231
Table 8-6.	L2 PLRU Bit Update Rules for Loads and Stores	231
Table 8-7.	L2 PLRU Bit Update Rules for Cache Operations	231
Table 8-8.	L1 - L2 Combined Coherency State	232
Table 8-9.	L2 Data Cache State Transitions in Response to L1/LSU Operations	232
Table 9-1.	Performance Monitor SPRs	244
Table 9-2.	PMC1 Events—MMCR0[19:25] Select Encodings	248
Table 9-3.	PMC2 Events—MMCR0[26:31] Select Encodings	249
Table 9-4.	PMC3 Events—MMCR1[0:4] Select Encodings	250
Table 9-5.	PMC4 Events—MMCR1[5:9] Select Encodings	251
Table 10-1.	Split-Field Notation and Conventions	255
Table 10-2.	Instruction Syntax Conventions	256
Table 10-3.	Notation and Conventions	257
Table 10-4.	Instruction Field Conventions	259
Table 10-5.	Precedence Rules	260
Table 10-6.	BO Operand Encodings	276
Table 10-7.	BO Operand Encodings	278
Table 10-8.	BO Operand Encodings	280
Table 10-9.	Operand Values (fresx)	334
Table 10-10.	Operand Values (frsqrtext)	337
Table 10-11.	TBR Encodings for mftb	383
Table A-1.	Complete Instruction List Sorted by Opcode	495
Table A-2.	Integer Arithmetic Instructions	502

Table A-3.	Integer Compare Instructions	503
Table A-4.	Integer Logical Instructions	503
Table A-5.	Integer Rotate Instructions	503
Table A-6.	Integer Shift Instructions	504
Table A-7.	Floating-Point Arithmetic Instructions	504
Table A-8.	Floating-Point Multiply-Add Instructions	504
Table A-9.	Floating-Point Rounding and Conversion Instructions	505
Table A-10.	Floating-Point Compare Instructions	505
Table A-11.	Floating-Point Status and Control Register Instructions	505
Table A-12.	Integer Load Instructions	506
Table A-13.	Integer Store Instructions	506
Table A-14.	Integer Load and Store with Byte Reverse Instructions	507
Table A-15.	Integer Load and Store Multiple Instructions	507
Table A-16.	Integer Load and Store String Instructions	507
Table A-17.	Memory Synchronization Instructions	507
Table A-18.	Floating-Point Load Instructions	508
Table A-19.	Floating-Point Store Instructions	508
Table A-20.	Floating-Point Move Instructions	508
Table A-21.	Branch Instructions	509
Table A-22.	Condition Register Logical Instructions	509
Table A-23.	System Linkage Instructions	509
Table A-24.	Trap Instructions	509
Table A-25.	Processor Control Instructions	510
Table A-26.	Cache Management Instructions	510
Table A-27.	Segment Register Manipulation Instructions	510
Table A-28.	Lookaside Buffer Management Instructions	511
Table A-29.	External Control Instructions	511
Table A-30.	Paired-Single Load and Store Instructions	511
Table A-31.	Paired-Single Floating-Point Arithmetic Instructions	512
Table A-32.	Miscellaneous Paired-Single Instructions	512



Revision Log

Each release of this document supersedes all previously released versions. The revision log lists all significant changes made to the document since its initial release. In the rest of the document, change bars in the margin indicate that the adjacent text was modified from the previous release of this document.

Revision Date	Description
September 28, 2011	Version 0.2.
June 22, 2011	Version 0.1 (initial developer's user manual version).



Preface

This document defines the functionality of the IBM® PowerPC® Espresso microprocessor for software and hardware developers.

The information in this book is subject to change without notice, as described in the disclaimers on the legal page. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation. To locate any published errata or updates for this document, contact your IBM representative.

Audience

This manual is intended for system software and hardware developers and application programmers who want to develop products for the Espresso microprocessor. It is assumed that the reader understands operating systems, microprocessor system design, basic principles of reduced instruction set computer (RISC) processing, and details of the PowerPC Architecture.

Organization

For ease in reference, the arrangement of topics in this book is similar to that of the *PowerPC Microprocessor Family: The Programming Environments Manual for 32-Bit Microprocessors*. Topics build upon one another, beginning with a description and summary of Espresso-specific registers and instructions and progressing to more specialized topics such as Espresso-specific details regarding the cache, exception, memory management models, and power management. Thus, chapters might include information from multiple levels of the architecture. For example, the discussion of the cache model uses information from both the virtual environment architecture (VEA) and the operating environment architecture (OEA).

A summary and a brief description of the major sections of this manual follows:

- *Chapter 1. "Espresso Overview"* is useful for readers who want a general understanding of the features and functions of the PowerPC Architecture and the Espresso cores and processor. This chapter describes the flexible nature of the PowerPC Architecture definition, and provides an overview of how the PowerPC Architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, exception model, and memory management model.
- *Chapter 2. "Programming Model"* is useful for software engineers who need to understand the Espresso-specific registers, operand conventions, and details regarding how the PowerPC instructions are implemented on the Espresso microprocessor. Instructions are organized by function.
- *Chapter 3. "Espresso Instruction and Data Cache Operation"* describes the storage subsystem as implemented on the Espresso microprocessor. The storage subsystem includes the core interface logic, the noncacheable unit, the L2 cache and controls, and the bus interface unit.
- *Chapter 4. "Exceptions"* describes the exception model defined in the PowerPC OEA and the specific exception model implemented on the Espresso microprocessor.
- *Chapter 5. "Memory Management"* describes the Espresso implementation of the memory management unit specifications provided by the PowerPC OEA for PowerPC processors.
- *Chapter 6. "Instruction Timing"* describes key design characteristics of the Espresso microprocessor.

- *Chapter 7. "Memory Subsystem Operation"* describes the processor interface (PI), which is a bus architecture providing high-speed, high-performance interconnections for processors, I/O devices, memory subsystems, and bridge chips.
- *Chapter 8. "L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe"* describes the Espresso microprocessor implementation of the L2 cache, L1 D-Cache partition, direct memory access (DMA), and write gather pipe.
- *Chapter 9. "Performance Monitor"* describes the operation of the performance monitor diagnostic tool incorporated in the Espresso microprocessor and provides detailed event information.
- *Chapter 10. "Espresso PowerPC Instruction Set"* lists the PowerPC instruction set in alphabetical order by mnemonic.

Related Documents

This manual is intended as a companion to the following document:

- *PowerPC Microprocessor Family: Programming Environments Manual for 32-Bit Microprocessors* (referred to as the *Programming Environments Manual*). Provides information about resources defined by the PowerPC Architecture that are common to PowerPC processors. This manual describes the functionality of the 32-bit architecture model.

1. Espresso Overview

The IBM® Espresso RISC Microprocessor is an implementation of the PowerPC Architecture with enhancements to improve the floating-point performance and the data transfer capability. The Espresso microprocessor is a 3-core design and is based on the Broadway chip. The Broadway chip was derived from the earlier Gekko processor. This section provides an overview of the Espresso microprocessor features, including a block diagram showing the major functional components. It also provides information about how the Espresso implementation complies with the PowerPC Architecture definition.

Note: The Espresso microprocessor incorporates three complete Espresso-PowerPC cores on a single chip, along with some common logic to connect these cores to a system. The terms “Espresso core”, “Espresso microprocessor core”, and “core” are used interchangeably to describe each of the three cores. The term “core” refers to the instruction fetch and execution logic, including the level 1 (L1) cache, but excluding the storage subsystem, of each processor. The term “Espresso microprocessor” refers to the single chip module comprising the three cores, level 2 (L2) and the common logic. The term “Espresso” is used to refer to either the chip or the core when the context is clear.

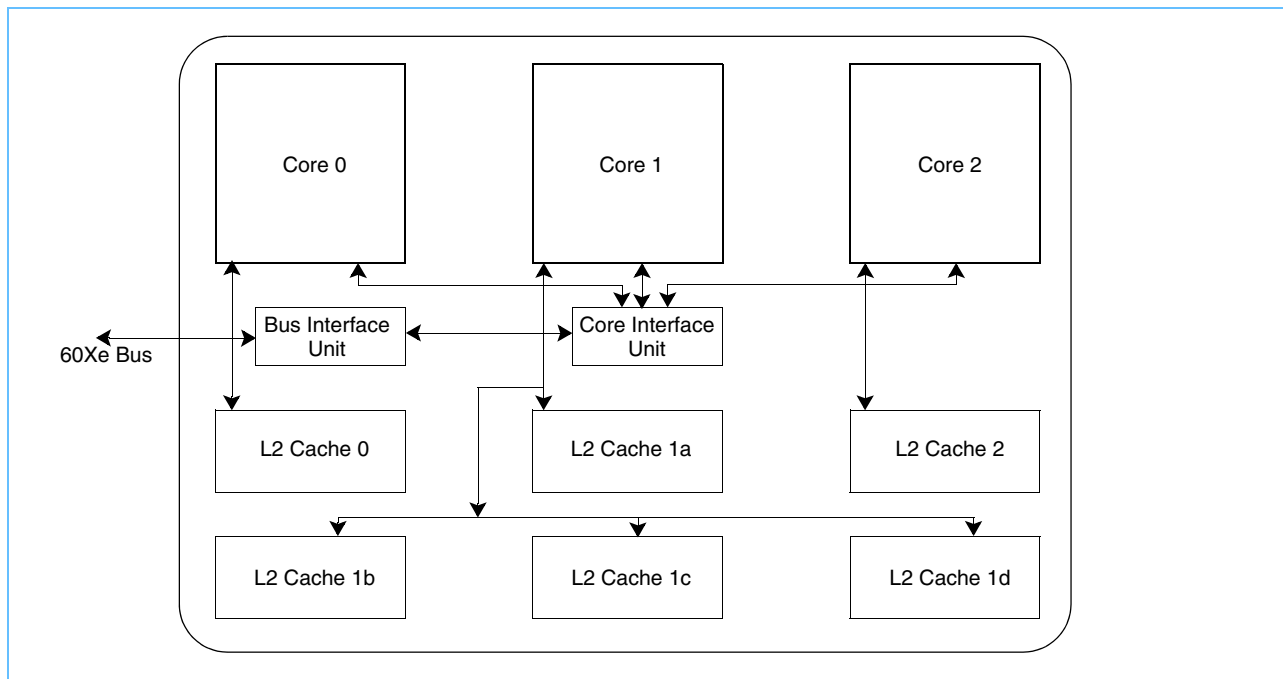
1.1 Espresso Microprocessor Overview

The Espresso microprocessor supports the modified, exclusive, recent, shared, invalid (MERSI) cache coherency protocol. While the previous design supported the modified, exclusive, invalid (MEI) cache coherency protocol, each Espresso core supports the MESI protocol. The R (recent) state is used to break ties when cores intervene. This state is not explicitly represented in the cores, but the core interface unit (CIU) provides functionality equivalent to that associated with the R state in routing intervention data to a requester.

Each of the three Espresso cores has its own private L2 cache. Two cores have a 512 KB, 4-way set-associative, 2-sectored cache, and one core has a 2 MB, 4-way, set-associative, 2-sectored cache. The Espresso cores use embedded dynamic random access memory (eDRAM) instead of SRAM for the L2 data arrays. See *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229.

Figure 1-1 on page 20 shows a block diagram of the Espresso microprocessor.

Figure 1-1. Espresso Microprocessor Block Diagram



1.1.1 On-Chip L2 Cache Implementation

The L2 cache is a unified cache that receives memory requests from both the L1 instruction and data caches independently. The L2 cache is implemented with an L2 Cache Control Register (L2CR), an on-chip, 4-way, set-associative tag array, and with a 512 KB or 2 MB eDRAM for data storage. The L2 cache normally operates in write-back mode and supports cache coherency through snooping. The access interface to the L2 is 64 bits and requires four cycles to read or write a single cache block.

The L2 cache is organized with 64-byte lines, which in turn are subdivided into 32-byte blocks, the unit at which cache coherency is maintained. This reduces the size of the tag array, because one tag can support two cache blocks. Each 32-byte cache block has its own coherency status bits. When a cache line is removed, both blocks and the tag are removed from the L2 cache. The cache block is only written to system memory if the modified bit is set.

Requests from the L1 cache generally result from instruction misses, data load or store misses, write-through operations, or cache management instructions. Misses from the L1 cache are looked up in the L2 tags and serviced by the L2 cache if they hit; they are forwarded to the core interface unit if they miss.

The L2 cache can accept multiple, simultaneous accesses; however, they are serialized and processed one at a time. The L1 instruction cache can request an instruction at the same time that the L1 data cache is requesting one load and two store operations. The L2 cache also services snoop requests from the other processors. If there are multiple pending requests to the L2 cache, snoop requests have highest priority. The next priority consists of load and store requests from the L1 data cache. The next priority consists of instruction fetch requests from the L1 instruction cache. A load miss normally results in a request for the 32-byte sector containing the required instruction or data. Optionally, the L2 cache can be configured to request a 64-byte line.



1.1.2 Core Interface Unit

The core interface unit (CIU) accepts load and store requests, as well as coherency operations, from the three cores, arbitrates among those requests, broadcasts those requests to the other cores to maintain coherency and for possible intervention, and forwards the requests to the bus interface unit (BIU) as necessary. The arbitration scheme is round robin, with an additional feature to throttle requests from individual cores to allow software control of bandwidth allocation. See *Section 7.2 Core Interface Unit* on page 222 for more information.

1.2 Espresso Core Features

This section describes the features and general operation of the Espresso core. The interrelationship of these features is shown in *Figure 1-2 Espresso Core Block Diagram*. The Espresso core is an implementation of the PowerPC microprocessor family of reduced instruction set computer (RISC) microprocessors with extensions to improve the floating-point performance. The Espresso core implements the 32-bit portion of the PowerPC Architecture, which provides 32-bit effective addresses, integer data types of 8, 16, and 32 bits, and floating-point data types of single-precision and double-precision. The Espresso core extends the PowerPC Architecture with the paired single-precision floating-point data type and a set of paired single floating-point instructions. Espresso is a superscalar processor that can complete two instructions simultaneously. It incorporates the following six execution units:

- Floating-point unit (FPU)
- Branch processing unit (BPU)
- System register unit (SRU)
- Load/store unit (LSU)
- Two integer units (IUs): IU1 executes all integer instructions. IU2 executes all integer instructions except multiply and divide instructions.

The ability to execute several instructions in parallel and the use of simple instructions with rapid execution times yield high efficiency and throughput for Espresso-based systems. Most integer instructions execute in one clock cycle. The FPU is pipelined; it breaks the tasks it performs into subtasks, and then executes in three successive stages. Typically, a floating-point instruction can occupy only one of the three stages at a time, freeing the previous stage to work on the next floating-point instruction. Thus, three single or paired single-precision floating-point instructions can be in the FPU execute stage at a time. Double-precision add instructions have a 3-cycle latency; double-precision multiply and multiply-add instructions have a 4-cycle latency.

Figure 1-2 Espresso Core Block Diagram on page 23 shows the parallel organization of the execution units, which are shaded in the diagram. The instruction unit fetches, dispatches, and predicts branch instructions. Note that this is a conceptual model that shows basic features rather than attempting to show how features are implemented physically.

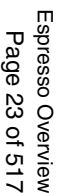
The Espresso core has an independent on-chip, 32 KB, 8-way set-associative, physically addressed L1 cache for instructions and data, and independent instruction and data memory management units (MMUs). The data cache can be configured as a 4-way, 16 KB locked cache or a 4-way, 16 KB normal cache. Each MMU has a 128-entry, 2-way set-associative translation lookaside buffer (DTLB and ITLB) that saves recently used page address translations. Block address translation is done through the 8-entry instruction and data block address translation (IBAT and DBAT) arrays, defined by the PowerPC Architecture. During block translation, effective addresses are compared simultaneously with all BAT entries.

For information about the L1 cache, see *Section 3 Espresso Instruction and Data Cache Operation* on page 111.

The Espresso core has a direct memory access (DMA) engine to transfer data from the external memory to the locked data cache and to transfer data from the locked data cache to the external memory.

A write gather pipe is implemented for efficient noncacheable store operations.

Version 0.2
September 28, 2011—IBM Confidential—Available under NDA only



1.2.1 Overview of Espresso Core Features

Major features of the Espresso core are as follows.

- High-performance, superscalar microprocessor.
 - As many as four instructions can be fetched from the instruction cache per clock cycle.
 - As many as two instructions can be dispatched per clock.
 - As many as six instructions can execute per clock (including two integer instructions).
 - Single-clock-cycle execution for most instructions.
- Six independent execution units and two register files.
 - BPU featuring both static and dynamic branch prediction.
 - 64-entry (16-set, 4-way set-associative) branch target instruction cache (BTIC), a cache of branch instructions that have been encountered in branch/loop code sequences. If a target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically, if a fetch access hits the BTIC, it provides the first two instructions in the target stream, effectively yielding a zero cycle branch.
 - 512-entry branch history table (BHT) with 2 bits per entry for four levels of prediction—not-taken, strongly not-taken, taken, strongly taken.
 - Branch instructions that do not update the count register (CTR) or link register (LR) are removed from the instruction stream.
 - Two integer units (IUs) that share 32 GPRs for integer operands.
 - IU1 can execute any integer instruction.
 - IU2 can execute all integer instructions except multiply and divide instructions (multiply, divide, shift, rotate, arithmetic, and logical instructions). Most instructions that execute in the IU2 take one cycle to execute. The IU2 has a single-entry reservation station.
 - Three-stage FPU.
 - Supports paired single-precision, floating-point arithmetic instruction set extension.
 - Fully IEEE 754-1985-compliant FPU for both single-precision and double-precision operations.
 - Supports non-IEEE mode for time-critical operations.
 - Hardware support for denormalized numbers.
 - Two-entry reservation station.
 - Thirty-two 64-bit FPRs for single-precision, paired single-precision, or double-precision operands.
 - Two-stage LSU.
 - Two-entry reservation station.
 - Single-cycle, pipelined cache access.
 - Dedicated adder performs effective address (EA) calculations.
 - Performs alignment and precision conversion for floating-point data.
 - Performs alignment and sign extension for integer data.
 - Three-entry store queue.

- Supports both big-endian and little-endian modes.
- Supports data type conversion with indexed scaling.
- SRU handles miscellaneous instructions.
 - Executes Condition Register (CR) logical and Move To/Move From Special Purpose Register (SPR) instructions (**mtspr** and **mfspr**).
 - Single-entry reservation station.
- Rename buffers.
 - Six general purpose register (GPR) rename buffers.
 - Six floating-point register (FPR) rename buffers.
 - Condition register buffering supports two CR writes per clock.
- Completion unit.
 - The completion unit retires an instruction from the 6-entry reorder buffer (completion queue) when all instructions ahead of it have been completed, the instruction has finished execution, and no exceptions are pending.
 - Guarantees a sequential programming model and a precise exception model.
 - Monitors all dispatched instructions and retires them in order.
 - Tracks unresolved branches and flushes instructions from the mispredicted branch path.
 - Retires as many as two instructions per clock.
- Separate on-chip L1 instruction and data caches (Harvard architecture).
 - 32 KB, 8-way set-associative instruction and data caches.
 - Pseudo least-recently-used (PLRU) replacement algorithm.
 - 32-byte (8-word) cache block.
 - Physically indexed/physical tags. (Note that the PowerPC Architecture refers to physical address space as real address space.)
 - Cache write-back or write-through operation programmable on a virtual page or BAT block basis.
 - Instruction cache can provide four instructions per clock; data cache can provide two words per clock
 - Caches can be disabled in software
 - Caches can be locked in software
 - Data cache coherency (MESI) is maintained in hardware
 - The critical doubleword is made available to the requesting unit when it is read into the line-fill buffer. The cache is nonblocking, so it can be accessed during this block reload.
 - The data cache can be partitioned as a 4-way, 16 KB normal cache or a 4-way, 16 KB locked cache.
- L2 cache controller support for either a 256 KB, 512 KB, or 2 MB L2 cache.
- DMA engine.
 - 15-entry DMA command queue.
 - Each DMA command can transfer up to 4 KB data in 32-byte increments.

- User mode DMA
 - Extension to the DMA facility to support user-mode access.
- Write gather pipe.
 - 128-byte circular first-in first-out (FIFO) buffer.
 - Noncacheable stores to a specified address are gathered for a burst transaction transfer.
 - Partial lines can be saved during a context switch.
- Separate memory management units (MMUs) for instructions and data.
 - 52-bit virtual address; 32-bit physical address.
 - Address translation for virtual pages or variable-sized BAT blocks.
 - Memory programmable as write-back or write-through, cacheable or noncacheable, and coherency enforced or coherency not enforced on a virtual page or BAT block basis.
 - Eight separate IBAT arrays for instructions and eight separate DBAT arrays for data.
 - Separate virtual instruction and data translation lookaside buffers (TLB).
 - Both TLBs are 128-entry, 2-way set associative and use the LRU replacement algorithm.
 - TLBs are hardware-reloadable (the page table search is performed by hardware).
 - Page size configurable to either 4 KB or 128 KB.
- Memory interface features include the following.
 - A 64-bit internal data bus with burst transfers.
 - Eight-word reload buffer for the L1 data cache.
 - Single-entry load queue.
 - Single-entry instruction fetch queue.
 - Two-entry L2 cache castout queue.
 - Single-entry snoop queue.
- Multiprocessing support features include the following:
 - Hardware-enforced, 4-state cache-coherency protocol (modified, exclusive, shared, invalid [MESI]) for the data cache.
 - Load/store with reservation instruction pair for atomic memory references, semaphores, and other multiprocessor operations
- Power and thermal management
 - Doze mode to reduce power dissipation when the core is idle.

Note: All the functional units are disabled except for the time base/decrementer registers and the snooping logic.
 - Instruction cache throttling provides control to limit power consumption by slowing instruction fetching.
 - Low-power functional mode achieved through frequency and voltage reduction.
- The performance monitor can be used to help debug system designs and improve software efficiency.
- In-system testability and debugging features through JTAG boundary-scan capability.

1.2.2 Instruction Flow

As shown in *Figure 1-2 Espresso Core Block Diagram* on page 23, the Espresso instruction unit provides centralized control of instruction flow to the execution units. The instruction unit contains a sequential fetcher, a 6-entry instruction queue (IQ), a dispatch unit, and a BPU. It determines the address of the next instruction to be fetched based on information from the sequential fetcher and from the BPU. See *Section 6 Instruction Timing* on page 183 for more information.

The sequential fetcher loads instructions from the instruction cache into the instruction queue. The BPU extracts branch instructions from the sequential fetcher. Branch instructions that cannot be resolved immediately are predicted using either Espresso-specific dynamic branch prediction or the architecture-defined static branch prediction.

Branch instructions that do not update the LR or CTR are removed from (folded out of) the instruction stream. Instruction fetching continues along the predicted path of the branch instruction.

Instructions issued to execution units beyond a predicted branch can be executed but are not retired until the branch is resolved. If branch prediction is incorrect, the completion unit flushes all instructions fetched on the predicted path, and instruction fetching resumes along the correct path.

1.2.2.1 Instruction Queue and Dispatch Unit

The instruction queue (IQ), shown in *Figure 1-2 Espresso Core Block Diagram* on page 23, holds as many as six instructions and loads up to four instructions from the instruction cache during a single processor clock cycle. The instruction fetcher continuously attempts to load as many instructions as there were vacancies created in the IQ in the previous clock cycle. All instructions except branches are dispatched to their respective execution units from the bottom two positions in the instruction queue (IQ0 and IQ1) at a maximum rate of two instructions per cycle. Reservation stations are provided for the IU1, IU2, FPU, LSU, and SRU for dispatched instructions. The dispatch unit checks for source and destination register dependencies, allocates rename buffers, determines whether a position is available in the completion queue, and inhibits subsequent instruction dispatching if these resources are not available.

Branch instructions can be detected, decoded, and predicted from anywhere in the instruction queue. For a more detailed discussion of instruction dispatch, see *Section 6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements* on page 210.

1.2.2.2 Branch Processing Unit

The BPU receives branch instructions from the sequential fetcher and performs CR lookahead operations on conditional branches to resolve them early, achieving the effect of a zero-cycle branch in many cases.

Unconditional branch instructions and conditional branch instructions in which the condition is known can be resolved immediately. For unresolved conditional branch instructions, the branch path is predicted using either the architecture-defined static branch prediction or Espresso-specific dynamic branch prediction. Dynamic branch prediction is enabled if `HID0[BHT] = '1'`.

When a prediction is made, instruction fetching, dispatching, and execution continue along the predicted path, but instructions cannot be retired and write results back to architected registers until the prediction is determined to be correct (resolved). When a prediction is incorrect, the instructions from the incorrect path are flushed from the processor and instruction fetching resumes along the correct path. The Espresso core allows a second branch instruction to be predicted; instructions from the second predicted branch instruction stream can be fetched but cannot be dispatched. These instructions are held in the instruction queue.

Dynamic prediction is implemented using a 512-entry branch history table (BHT), a cache that provides two bits per entry that together indicate four levels of prediction for a branch instruction: not-taken, strongly not-taken, taken, strongly taken. When dynamic branch prediction is disabled, the BPU uses a bit in the instruction encoding to predict the direction of the conditional branch. Therefore, when an unresolved conditional branch instruction is encountered, Espresso executes instructions from the predicted path although the results are not committed to architected registers until the conditional branch is resolved. This execution can continue until a second unresolved branch instruction is encountered.

When a branch is taken (or predicted as taken), the instructions from the untaken path must be flushed and the target instruction stream must be fetched into the IQ. The BTIC is a 64-entry cache that contains the most recently used branch target instructions, typically in pairs. When an instruction fetch hits in the BTIC, the instructions arrive in the instruction queue in the next clock cycle, a clock cycle sooner than they would arrive from the instruction cache. Additional instructions arrive from the instruction cache in the next clock cycle. The BTIC reduces the number of missed opportunities to dispatch instructions and gives the processor a one-cycle head start on processing the target stream. With the use of the BTIC, the Espresso core achieves a zero cycle delay for branches taken. Coherency of the BTIC table is maintained by a table reset on an instruction cache (I-Cache) flush invalidate, **icbi** or **rfi** instruction execution, or when an exception is taken.

The BPU contains an adder to compute branch target addresses and three user-control registers: the Link Register (LR), the Count Register (CTR), and the CR. The BPU calculates the return pointer for subroutine calls and saves it into the LR for certain types of branch instructions. The LR also contains the branch target address for the Branch Conditional to Link Register (**bclrx**) instruction. The CTR contains the branch target address for the Branch Conditional to Count Register (**bcctrx**) instruction. Because the LR and CTR are SPRs, their contents can be copied to or from any GPR. Because the BPU uses dedicated registers rather than GPRs or FPRs, execution of branch instructions is largely independent from execution of integer and floating-point instructions.

1.2.2.3 Completion Unit

The completion unit operates closely with the dispatch unit. Instructions are fetched and dispatched in program order. At the point of dispatch, the program order is maintained by assigning each dispatched instruction a successive entry in the 6-entry completion queue. The completion unit tracks instructions from dispatch through execution and retires them in program order from the two bottom entries in the completion queue (CQ0 and CQ1).

Instructions cannot be dispatched to an execution unit unless there is a vacancy in the completion queue and rename buffers are available. Branch instructions that do not update the CTR or LR are removed from the instruction stream and do not occupy a space in the completion queue. Instructions that update the CTR and LR follow the same dispatch and completion procedures as nonbranch instructions, except that they are not issued to an execution unit.

An instruction is retired when it is removed from the completion queue and its results are written to architected registers (GPRs, FPRs, LR, and CTR) from the rename buffers. In-order completion ensures program integrity and the correct architectural state when the Espresso core must recover from a mispredicted branch or any exception. Also, the rename buffers assigned to it by the dispatch unit are returned to the available rename buffer pool. These rename buffers are reused by the dispatch unit for subsequent instructions being dispatched.

For a more detailed discussion of instruction completion, see *Section 6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements* on page 210.

1.2.2.4 Independent Execution Units

In addition to the BPU, the Espresso core has the following five execution units:

- Two Integer Units (IUs)
- Floating-Point Unit (FPU)
- Load/Store Unit (LSU)
- System Register Unit (SRU)

Each unit is described in the following sections.

Integer Units (IUs)

The IU1 and IU2 integer units are shown in *Figure 1-2 Espresso Core Block Diagram* on page 23. The IU1 can execute any integer instruction; the IU2 can execute any integer instruction except multiplication and division instructions. Each IU has a single-entry reservation station that can receive instructions from the dispatch unit and operands from the GPRs or the rename buffers. The output of the IU is latched in the rename buffer assigned to the instruction by the dispatch unit.

Each IU consists of three single-cycle subunits: a fast adder/comparator, a subunit for logical operations, and a subunit for performing rotates, shifts, and count-leading-zero operations. These subunits handle all single-cycle arithmetic and logical integer instructions; only one subunit can execute an instruction at a time.

The IU1 has a 32-bit integer multiplier/divider as well as the adder, shift, and logical units of the IU2. The multiplier supports early exit for operations that do not require full 32×32 bit multiplication. Multiply and divide instructions spend several cycles in the execution stage before the results are written to the output rename buffer.

Floating-Point Unit (FPU)

The FPU, shown in *Figure 1-2 Espresso Core Block Diagram* on page 23, is designed as a 3-stage pipelined processing unit, where the first stage is for multiply, the second stage is for add, and the third stage is for normalize. A single-precision multiply-add operation is processed with 1-cycle throughput and 3-cycle latency (a single-precision instruction spends 1-cycle in each stage of the FPU). A double-precision multiply requires two cycles in the multiply stage and 1-cycle in each additional stage. A double-precision multiply-add has a 2-cycle throughput and a 4-cycle latency. As instructions are dispatched to the FPU reservation station, source operand data can be accessed from the FPRs or from the FPR rename buffers. Results in turn are written to the rename buffers and are made available to subsequent instructions. Instructions pass through the reservation station and the pipeline stages in program order. Stalls due to contention for FPRs are minimized by automatic allocation of the six floating-point rename buffers. The completion unit writes the contents of the rename buffer to the appropriate FPR when floating-point instructions are retired.

The Espresso core supports all IEEE 754 floating-point data types (normalized, denormalized, not-a-number [NaN], zero, and infinity) in hardware, eliminating the latency incurred by software exception routines. (Note that “exception” is also referred to as “interrupt” in the architecture specification.) For paired single-precision operations, both data paths comply with the IEEE standard independently.

Load/Store Unit (LSU)

The LSU executes all load and store instructions and provides the data transfer interface between the GPRs, FPRs, and the data cache/memory subsystem. The LSU functions as a 2-stage pipelined unit where it calculates effective addresses in the first stage. In the second stage, the address is translated, the cache is accessed, and the data is aligned if necessary. Unless extensive data alignment is required (for example, crossing a doubleword boundary), the instructions complete in two cycles with a 1-cycle throughput. The LSU also provides sequencing for load/store string and multiple register transfer instructions.

The Espresso core implements eight paired-single quantization load and store instructions. The load instructions read a pair of 8-bit or 16-bit, signed or unsigned integers, convert them into single-precision floating-point data with the scaling factor in the quantization register, and write the results into the FPR. The store instructions read the 64-bit data from the FPR as paired single-precision floating-point data, convert the single-precision floating-point numbers into paired, 8-bit or 16-bit, signed or unsigned integer data, and store the results.

Load and store instructions are translated and issued in program order; however, some memory accesses can occur out of order. Synchronizing instructions can be used to enforce strict ordering if necessary. When there are no data dependencies and the guard bit for the page or block is cleared, a maximum of one out-of-order cacheable load operation can execute per cycle, with a 2-cycle total latency on a cache hit. Data returned from the cache is held in a rename buffer until the completion logic commits the value to a GPR or FPR. Stores cannot be executed out of order and are held in the store queue until the completion logic signals that the store operation is to be completed to memory. The Espresso core executes store instructions with a maximum throughput of one per cycle and a 3-cycle total latency to the data cache. The time required to perform the actual load or store operation depends on the processor/bus clock ratio and whether the operation involves the L1 cache, the L2 cache, system memory, or an I/O device.

System Register Unit (SRU)

The SRU executes various system-level instructions, as well as Condition Register logical operations and Move To/From Special-Purpose Register instructions. To maintain the system state, most instructions executed by the SRU are execution serialized with other instructions; that is, the instruction is held for execution in the SRU until all previously issued instructions have been retired. Results from execution-serialized instructions executed by the SRU are not available or forwarded for subsequent instructions until the instruction completes.

1.2.3 Memory Management Units (MMUs)

The Espresso MMUs support up to 4 petabytes (2^{52}) of virtual memory and 4 gigabytes (2^{32}) of physical memory for instructions and data. The MMUs also control access privileges for these spaces on block and page granularities. Referenced and changed status is maintained by the processor for each page to support demand-paged virtual memory systems.

The LSU, with the aid of the MMU, translates effective addresses for data loads and stores; the effective address is calculated on the first cycle, and the MMU translates it to a physical address at the same time it is accessing the L1 cache on the second cycle. The MMU also provides the necessary control and protection information to complete the access. By the end of the second cycle, the data and control information is available if no miss conditions for translation and cache access were encountered. This yields a 1-cycle throughput and a 2-cycle latency.

The Espresso core supports the following types of memory translation.

- **Real addressing mode:** In this mode, translation is disabled (control bits MSR[IR] = '0' for instructions and MSR[DR] = '0' for data), and the effective address is used as the physical address to access memory.
- **Virtual page address translation:** Translates the effective address into a physical address by using the segment registers and the TLB and accesses data from a 4 KB or 128 KB virtual page. This page is either in physical memory or on disk. If the latter, a page-fault exception occurs.
- **Block address translation:** Translates the effective address into a physical address by using the BAT registers and accesses a block (128 KB to 256 MB) in memory.

If translation is enabled, the appropriate MMU translates the higher-order bits of the effective address into physical address bits by either BATs or page translation method. The lower-order address bits (that are untranslated and therefore, considered both logical and physical) are directed to the L1 caches where they form the index into the 8-way set-associative tag and data arrays. After translating the address, the MMU passes the higher-order physical address bits to the cache, and the cache lookup completes. For caching-inhibited accesses or accesses that miss in the cache, the untranslated lower-order address bits are concatenated with the translated higher-order address bits; the resulting 32-bit physical address is used to access the L2 cache or system memory using the 60Xe bus.

If the BAT registers are enabled and the address is translated using this method, the page translation is canceled and the high-order physical address bits from the BAT register are forwarded to the cache/memory access system. There are eight 8-byte BAT registers for address translation. These registers provide cache control and protection information as well as address translation. Only one of the BAT entries should translate a given effective address.

If address relocation is enabled and the effective address is not translated using the BAT method, the virtual page method is used. The 4 high-order bits of the effective address are used to access the 16-entry segment register array. From this array, a 24-bit segment register is accessed and used to form the high-order bits of a 52-bit virtual address. The low-order 28-bits of the effective address are used to form the low-order bits of the virtual address. This 52-bit virtual address is translated into a physical address by doing a lookup in the TLB. If the lookup is successful, a physical address is formed by using 16 low-order bits from the virtual address and 16 high-order bits from the TLB. The TLB also provides cache control and protection information to be used by the cache/memory system.

The TLBs are 128-entry, 2-way set-associative caches that contain information about recently translated virtual addresses. When an address translation is not in a TLB, the Espresso core automatically generates a page table search in memory to update the TLB. This search might find the required entry in the L1 or L2 cache or in the page table in memory. The time to reload a TLB entry depends on where it is found and can be completed in just several cycles. If memory is searched, a maximum of 16 bus cycles are needed before a page fault exception is signaled.

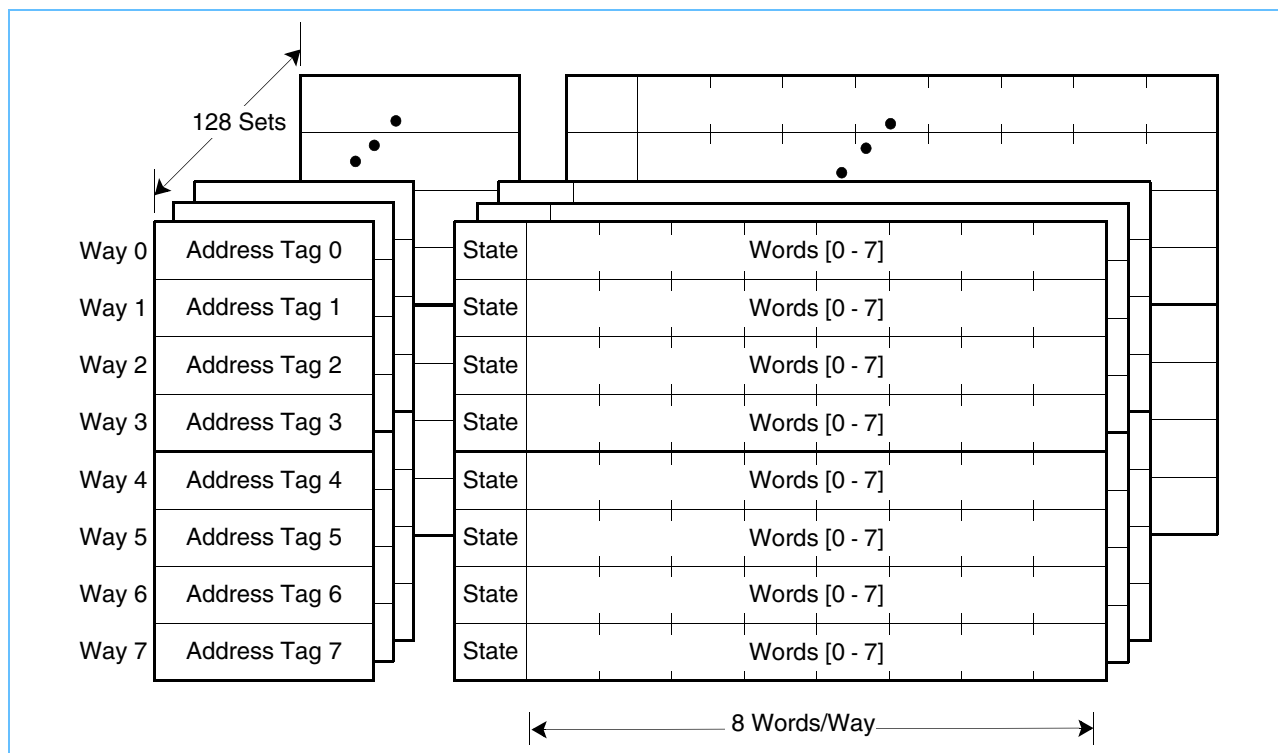
1.2.4 On-Chip L1 Instruction and Data Caches

The Espresso core implements separate instruction and data caches. Each cache is 32 KB and 8-way set associative. As defined by the PowerPC Architecture, they are physically indexed. Each cache block contains 8 contiguous words from memory that are loaded from an 8-word boundary (that is, bits EA[27:31] are zeros); thus, a cache block never crosses a page boundary. A miss in the L1 cache causes a block reload from either the L2 cache if the block is in the L2 cache or from main memory. The critical doubleword is accessed first and forwarded to the load/store unit and written into an 8-word buffer. Subsequent doublewords are fetched from either the L2 cache or the system memory and written into the buffer. Once the total block is in the buffer, the line is written into the L1 cache in a single cycle using a 256-bit buffer-to-L1 bus. This minimizes

write cycles into the L1 cache leaving more read/write cycles available to the LSU. The L1 cache is nonblocking and supports hits under misses during this reload. Misaligned accesses across a block or page boundary can incur a performance penalty.

The Espresso L1 cache organization is shown in *Figure 1-3*.

Figure 1-3. Cache Organization



The data cache provides doubleword access to the LSU in each cycle. Like the instruction cache, the data cache can be invalidated all at once or on a per-cache-block basis. The data cache can be disabled and invalidated by clearing `HID0[DCE]` and setting `HID0[DCFI]`. The data cache can be locked by setting `HID0[DLOCK]`. To ensure cache coherency, the data cache supports the 4-state MESI protocol. The data cache tags are single-ported, so that a simultaneous load or store and a snoop access represent a resource collision and an LSU access is delayed for one cycle. If a snoop hit occurs and a cast-out is required, the LSU is blocked internally for one cycle to allow the 8-word block of data to be copied to the write-back buffer.

The data bus width for MIU accesses of the L1 data cache array is 256 bits. Cache blocks can be read from or written to the cache array in a single cycle to minimize cache contention between the MIU, L1 cache, and the load/store unit.

By setting `HID2[LCE] = '1'`, the data cache can be configured into two partitions. The first partition, consisting of ways 0 - 3, forms a 16 KB normal data cache. The second partition, consisting of ways 4 - 7, forms a 16 KB locked cache that can be used as an on-chip memory. The detailed operation is defined in *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229. Within one cycle, the instruction cache provides up to four instructions to the instruction queue. The instruction cache can be invalidated entirely or on a cache-block basis. The instruction cache can be disabled and invalidated by clearing `HID0[ICE]` and setting `HID0[ICFI]`. The instruction cache can be locked by setting `HID0[ILOCK]`. The instruction cache supports only the valid/invalid states.

The Espresso core also implements a 64-entry (16-set, 4-way set-associative) BTIC. The BTIC is a cache of branch instructions that have been encountered in branch/loop code sequences. If the target instruction is in the BTIC, it is fetched into the instruction queue a cycle sooner than it can be made available from the instruction cache. Typically the BTIC contains the first two instructions in the target stream. The BTIC can be disabled and invalidated through software.

Coherency of the BTIC is transparent to the running software and is coupled with various functions in the Espresso core. When the BTIC is enabled and loaded with instruction pairs to support a zero-cycle delay on branches taken, the table must be invalidated if the underlying program changes. This is also true for the I-Cache. The BTIC is reset on an instruction cache flush invalidate, an **icbi** or **rfi** instruction, and any exception.

For more information and timing examples showing cache hit and cache miss latencies, see *Section 6.3.2 Instruction Fetch Timing* on page 190.

1.3 Espresso Microprocessor Implementation

The PowerPC Architecture is derived from the IBM POWER® architecture (Performance Optimized with Enhanced RISC architecture). The PowerPC Architecture shares the benefits of the Power Architecture® optimized for single-chip implementations. The PowerPC Architecture design facilitates parallel instruction execution and is scalable to take advantage of future technological gains.

This section describes the PowerPC Architecture in general, and provides specific details about the implementation of the Espresso microprocessor as a low-power, 32-bit member of the PowerPC processor family. The structure of this section follows the organization of the user's manual; each subsection provides an overview of each section.

- Registers and programming model. *Section 1.4 PowerPC Registers and Programming Model* on page 35 describes the registers for the operating environment architecture common among PowerPC processors and describes the programming model. It also describes the registers that are unique to Espresso microprocessor. The information in this section is described more fully in *Section 2 Programming Model* on page 49.
- Instruction set and addressing modes. *Section 1.5 Instruction Set* on page 38 describes the PowerPC instruction set and addressing modes for the PowerPC operating environment architecture, defines the PowerPC instructions implemented in the Espresso microprocessor, and describes new instruction set extensions to improve the performance of single-precision floating-point operations and the capability of data transfer. The information in this section is described more fully in *Section 1.1 Espresso Microprocessor Overview* on page 19.
- Cache implementation. *Section 1.6 On-Chip Cache Implementation* on page 40 describes the cache model that is defined generally for PowerPC processors by the virtual environment architecture. It also provides specific details about the Espresso cache implementation.
- Exception model. *Section 1.7 Exception Model* on page 41 describes the exception model of the PowerPC operating environment architecture and the differences in the Espresso exception model. The information in this section is described more fully in *Section 4 Exceptions* on page 133.
- Memory management. *Section 1.8 Memory Management* on page 43 describes generally the conventions for memory management among the PowerPC processors. This section also describes the Espresso implementation of the 32-bit PowerPC memory management specification. The information in this section is described more fully in *Section 5 Memory Management* on page 151.

- Instruction timing. *Section 1.9 Instruction Timing* on page 44 provides a general description of the instruction timing provided by the superscalar, parallel execution supported by the PowerPC Architecture and the Espresso core. The information in this section is described more fully in *Section 6 Instruction Timing* on page 183.
- Performance monitor. *Section 1.10 Performance Monitor* on page 46 describes the performance monitor facility, which system designers can use to help bring up, debug, and optimize software performance. The information in this section is described more fully in *Section 9 Performance Monitor* on page 243.

The following sections summarize the features of the Espresso microprocessor, distinguishing those that are defined by the architecture from those that are unique to Espresso implementation.

The PowerPC Architecture consists of the following layers. Adherence to the PowerPC Architecture can be described in terms of which of the following levels of the architecture is implemented.

- PowerPC user instruction set architecture (UISA). Defines the base user-level instruction set, user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- PowerPC virtual environment architecture (VEA). Describes the memory model for a multiprocessor environment, defines cache control instructions, and describes other aspects of virtual environments. Implementations that conform to the VEA also adhere to the UISA, but might not necessarily adhere to the OEA.
- PowerPC operating environment architecture (OEA). Defines the memory management model, supervisor-level registers, synchronization requirements, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA.

The PowerPC Architecture allows a wide range of designs for such features as cache and system interface implementations. Espresso implementations support the three levels of the architecture described previously. For more information about the PowerPC Architecture, see the *PowerPC Microprocessor Family: The Programming Environments* manual.

1.4 PowerPC Registers and Programming Model

The PowerPC Architecture defines register-to-register operations for most computational instructions. Source operands for these instructions are accessed from the registers or are provided as immediate values embedded in the instruction itself. The 3-register instruction format allows specification of a target register distinct from the two source operands. Only load and store instructions transfer data between registers and memory.

PowerPC processors have two levels of privilege: supervisor mode of operation (typically used by the operating system) and user mode of operation (used by the application software; it is also called problem state). The programming models incorporate 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each PowerPC microprocessor also has its own unique set of hardware implementation-dependent (HID) registers.

While running in supervisor mode, the operating system is able to execute all instructions and access all registers defined in the PowerPC Architecture. In this mode, the operating system establishes all address translation and protection mechanisms, loads all processor state registers, and sets up all other control mechanisms defined on the Espresso microprocessor. While running in user mode (problem state), many of these registers and facilities are not accessible and any attempt to read or write these registers results in a program exception.

Figure 2-1 Programming Model (User Model–VEA and UISA)–Espresso Microprocessor Registers on page 50 shows all the Espresso registers available at the user and supervisor level. The numbers to the right of the SPRs indicate the number that is used in the syntax of the instruction operands to access the register. For more information, see *Section 2 Programming Model* on page 49.

The following tables summarize the PowerPC registers implemented in the Espresso core. *Table 1-1* describes the registers (excluding SPRs) defined by the architecture.

Table 1-1. Architecture-Defined Registers (Excluding SPRs)

Register	Level	Function
CR	User	The Condition Register (CR) consists of eight 4-bit fields that reflect the results of certain operations, such as move, integer and floating-point compare, arithmetic, and logical instructions, and provide a mechanism for testing and branching.
FPRs	User	The 32 floating-point registers (FPRs) serve as the data source or destination for floating-point instructions. These 64-bit registers can hold single, paired single, or double-precision floating-point values.
FPSCR	User	The Floating-Point Status and Control Register (FPSCR) contains the floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE-754 standard.
GPRs	User	The 32 general purpose registers (GPRs) contain the address and data arguments addressed from source or destination fields in integer instructions. Also floating-point load and store instructions use GPRs to address memory.
MSR	Supervisor	The Machine State Register (MSR) defines the processor state. Its contents are saved when an exception is taken and restored when exception handling completes. The Espresso core implements MSR[POW], (defined by the architecture as optional), which is used to enable the power management feature. The Espresso-specific MSR[PM] bit is used to mark a process for the performance monitor.
SR0 - SR15	Supervisor	The sixteen 32-bit segment registers (SRs) define the 4 GB space as sixteen 256 MB segments. The Espresso core implements segment registers as two arrays—a main array for data accesses and a shadow array for instruction accesses; see <i>Figure 1-2 Espresso Core Block Diagram</i> on page 23. Loading a segment entry with the Move to Segment Register (mtsr) instruction loads both arrays. The mfsr instruction reads the master register, shown as part of the data MMU in <i>Figure 1-2</i> .

The OEA defines numerous special-purpose registers that serve a variety of functions, such as providing controls, indicating status, configuring the processor, and performing special operations. During normal execution, a program can access the registers shown in *Figure 2-1* on page 50, depending on the program's access privilege (supervisor or user, determined by the privilege-level [PR] bit in the MSR). GPRs and FPRs are accessed through operands that are defined in the instructions. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit, as the part of the execution of an instruction. Some registers can be accessed both explicitly and implicitly.

In the Espresso implementation, all SPRs are 32 bits wide. *Table 1-2* describes the architecture-defined SPRs implemented by the Espresso core. See the *PowerPC Microprocessor Family: The Programming Environments* manual, for detailed descriptions of these registers. *Section 2.1 Espresso Processor Register Set* describes how these registers are implemented in the Espresso core, and describes features that the PowerPC Architecture defines as optional which are implemented on the Espresso core.

Table 1-2. Architecture-Defined SPRs Implemented

Register	Level	Function
LR	User	The Link Register (LR) can be used to provide the branch target address and to hold the return address after branch and link instructions.
BATs	Supervisor	The architecture defines 16 block address translation registers (BATs), which operate in pairs. The Espresso core supports an enhanced BAT facility with an additional 16 BAT registers. There are eight pairs of data BATs (DBATs) and eight pairs of instruction BATs (IBATs). BATs are used to define and configure blocks of memory.
CTR	User	The Count Register (CTR) is decremented and tested by branch-and-count instructions.
DABR	Supervisor	The optional Data Address Breakpoint Register (DABR) supports the data address breakpoint facility.
DAR	User	The Data Address Register (DAR) holds the address of an access after an alignment or data storage interrupt (DSI) exception.
DEC	Supervisor	The Decrementer Register (DEC) is a 32-bit decrementing counter that provides a way to schedule time delayed exceptions.
DSISR	User	The Data Storage Interrupt Status Register (DSISR) defines the cause of data access and alignment exceptions.
EAR	Supervisor	The External Access Register (EAR) controls access to the external access facility through the External Control In Word Indexed (eciwxx) and External Control Out Word Indexed (ecowxx) instructions.
PVR	Supervisor	The Processor Version Register (PVR) is a read-only register that identifies the processor version and revision level. This is a global register in the Espresso microprocessor, shared by all three cores.
SDR1	Supervisor	The Storage Description Register 1 (SDR1) specifies the page table address and size used in virtual-to-physical page address translation.
SRR0	Supervisor	The Machine Status Save and Restore Register 0 (SRR0) saves the address used for restarting an interrupted program when a Return from Interrupt (rfi) instruction executes (that is, exceptions).
SRR1	Supervisor	The Machine Status Save and Restore Register 1 (SRR1) is used to save machine status on exceptions and to restore machine status when an rfi instruction is executed.
SPRG0 - SPRG3	Supervisor	Software-use SPRs (SPRG0 - SPRG3) are provided for operating system use.
TB	User: read Supervisor: read/write	The Time Base Register (TB) is a 64-bit register that maintains the time and date variable. The TB consists of two 32-bit fields: Time Base Upper (TBU) and Time Base Lower (TBL). This is a global register in the Espresso microprocessor, shared by all three cores.
XER	User	The Fixed-Point Exception Register (XER) contains the summary overflow bit, integer carry bit, overflow bit, and a field specifying the number of bytes to be transferred by a Load String Word Indexed (lswx) or Store String Word Indexed (stswx) instruction.

Table 1-3 describes the SPRs in the Espresso core that are not defined by the PowerPC Architecture. Section 2.1.2 Espresso-Specific Registers gives detailed descriptions of these registers, including bit descriptions.

Table 1-3. Implementation-Specific Registers

Register	Level	Function
DMAL, DMAU	Supervisor, User	The DMA Upper (DMAU) and DMA Lower (DMAL) Registers are used to issue the DMA commands.
GQR0 - GQR7	Supervisor, User	The Graphics Quantization Registers (GQR0 - GQR7) are used to determine the scaling factor and data type conversion for the quantization load/store instructions.
HID2	User (R)	The Hardware Implementation Dependent Register 2 (HID2) enables the paired-single floating-point operations, L1 cache partition, write pipe and DMA, and controls the exceptions associated with the DMA and the locked cache operations.
IABR	Supervisor	The Instruction Address Breakpoint Register (IABR) supports instruction address breakpoint exceptions. It can hold an address to compare with instruction addresses in the IQ. An address match causes an instruction address breakpoint exception.
ICTC	Supervisor	The Instruction Cache Throttling Control Register (ICTC) has bits for controlling the interval at which instructions are fetched into the instruction buffer in the instruction unit. This helps control the Espresso core overall junction temperature.
MMCR0 - MMCR1	Supervisor	The Monitor Mode Control Registers (MMCR0 - MMCR1) are used to enable various performance monitoring interrupt functions. UMMCR0 - UMMCR1 provide user-level read access to MMCR0 - MMCR1.
PIR	Supervisor, User	The Processor ID Register (PIR) is a 32-bit, read-only register; its low-order 2 bits contain the processor ID. Note: When access is made in user mode, the register is termed UPIR.
PMC1 - PMC4	Supervisor	The Performance Monitor Counter Registers (PMC1 - PMC4) are used to count specified events. UPMC1 - UPMC4 provide user-level read access to these registers.
UMMCR0 - UMMCR1	User	The User Monitor Mode Control Registers (UMMCR0 - UMMCR1) provide user-level read access to MMCR0 - MMCR1.
UPMC1 - UPMC4	User	The User Performance Monitor Counter Registers (UPMC1 - UPMC4) provide user-level read access to PMC1 - PMC4.
USIA	User	The User Sampled Instruction Address Register (USIA) provides user-level read access to the SIA Register.
WPAR	User (R)	The Write Gather Pipe Address Register (WPAR) specifies the address of the noncacheable stores to be gathered for burst transfer.

Global registers are shared among the three Espresso cores. There is one copy of each global register on the chip. Each core can access any given global register using the **mtspr** and **mfspr** instructions. However, software is responsible for managing these shared resources consistently. It is a programming error for two cores to write to a given global register at the same time. All cores have read access to the global registers at all times. In addition to the three new global registers, the time base registers, TBU and TBL, are also shared by all three cores in the Espresso microprocessor. Table 1-4 describes the Espresso global registers. Section 2.1.2 Espresso-Specific Registers on page 55 gives detailed descriptions of these registers, including bit descriptions.

Table 1-4. Implementation-Specific Shared Registers

Register	Level	Function
TBU, TBL	Supervisor	The Time Base Registers (TBU, TBL) provide a 64-bit time of day value.

1.5 Instruction Set

All PowerPC instructions are encoded as single-word (32-bit) instructions. Instruction formats are consistent among all instruction types (the primary opcode is always six bits, the register operands always specified in the same bit fields in the instruction), permitting efficient decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplify instruction pipelining.

For more information, see *Section 2 Programming Model* on page 49.

1.5.1 PowerPC Instruction Set

The PowerPC instructions are divided into the following categories:

- Integer instructions. These include computational and logical instructions.
 - Integer arithmetic instructions
 - Integer compare instructions
 - Integer logical instructions
 - Integer rotate and shift instructions
- Floating-point instructions. These include floating-point computational instructions, as well as instructions that affect the FPSCR.
 - Floating-point arithmetic instructions
 - Floating-point multiply/add instructions
 - Floating-point rounding and conversion instructions
 - Floating-point compare instructions
 - Floating-point status and control instructions
- Load/store instructions. These include integer and floating-point load and store instructions.
 - Integer load and store instructions
 - Integer load and store multiple instructions
 - Floating-point load and store
 - Primitives used to construct atomic memory operations (**lwarx** and **stwcx.** instructions)
- Flow control instructions. These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
 - Branch and trap instructions
 - Condition Register logical instructions (sets conditions for branches)
 - System Call instructions
- Processor control instructions. These instructions are used to synchronize memory accesses and manage caches, TLBs, and the segment registers.
 - Move to/from SPR instructions
 - Move to/from MSR instructions
 - Synchronize (processor and memory system) instructions

- Instruction synchronize instructions
- Order loads and stores instructions
- Memory control instructions. These instructions provide control of caches, TLBs, and SRs.
 - Supervisor-level cache management instructions
 - User-level cache instructions
 - Segment register manipulation instructions
 - Translation lookaside buffer management instructions

This grouping does not indicate the execution unit that executes a particular instruction or group of instructions.

Integer instructions operate on byte, halfword, and word operands. Floating-point instructions operate on single-precision (one word) and double-precision (two word) floating-point operands. The PowerPC Architecture uses instructions that are 4 bytes long and word-aligned. It provides for integer byte, halfword, and word operand loads and stores between memory and a set of 32 GPRs. It also provides for single-precision and double-precision loads and stores between memory and a set of 32 floating-point registers (FPRs).

Computational instructions do not access memory. To use a memory operand in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written back to the target location using three or more instructions.

PowerPC processors follow the program flow when they are in the normal execution state; however, the flow of instructions can be interrupted directly by the execution of an instruction or by an asynchronous event. Either type of exception causes the associated exception handler to be invoked.

Effective address computations for both data and instruction accesses use 32-bit signed twos complement binary arithmetic. A carry from bit 0 and overflow are ignored.

1.5.2 Espresso Microprocessor Instruction Set

In addition to the 32-bit single-precision and the 64-bit double-precision floating-point operands, the Espresso microprocessor implements a new floating-point operand type: paired single-precision. The paired single operand uses a 64-bit FPR to maintain two 32-bit single-precision floating-point operands. As described in the following list, the PowerPC instruction set is substantially extended to support the paired single data type.

- The Espresso microprocessor provides hardware support for all 32-bit PowerPC instructions.
- The Espresso microprocessor implements the following instructions that are optional in the PowerPC Architecture:
 - External Control In Word Indexed (**eciwx**)
 - External Control Out Word Indexed (**ecowx**)
 - Floating Select (**fsel**)
 - Floating Reciprocal Estimate Single-Precision (**fres**)¹
 - Floating Reciprocal Square Root Estimate (**frsqrte**)¹
 - Store Floating-Point as Integer Word Indexed (**stfiwx**)

1. The **fres** and **frsqrte** instructions have a resolution of less than 1/4000.

- Translation Lookaside Buffer Invalidate Entry (**tlbie**)
- Translation Lookaside Buffer Synchronize (**tlbsync**)
- The Espresso microprocessor implements the Data Cache Block Zero and Lock (**dcbz_l**) instruction, which is not included in the PowerPC Architecture, to support the cache line allocation in the locked cache.
- The Espresso microprocessor implements the following instruction set extension, which is not included in the PowerPC Architecture, to support the paired single data type:
 - Quantization load instructions
 - Quantization store instructions
 - Floating-point instructions to support the paired single operand data type

1.6 On-Chip Cache Implementation

The following sections describe the PowerPC Architecture treatment of the cache and the Espresso implementation. A detailed description of the Espresso L1 cache implementation is provided in *Section 3 Espresso Instruction and Data Cache Operation* on page 111.

1.6.1 PowerPC Cache Model

The PowerPC Architecture does not define hardware aspects of cache implementations. For example, PowerPC processors can have unified caches, separate instruction and data caches (Harvard architecture), or no cache at all. PowerPC microprocessors control the following memory access modes on a virtual page or block (BAT) basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency

The caches are physically addressed, and the data cache can operate in either write-back or write-through mode, as specified by the PowerPC Architecture.

The PowerPC Architecture defines the term “cache block” as the cacheable unit. The VEA and OEA define cache management instructions that a programmer can use to affect cache contents.

1.6.2 Espresso Microprocessor Cache Implementation

Espresso cache implementation is described in *Section 1.2.4 On-Chip L1 Instruction and Data Caches* on page 31 and *Section 1.1.1 On-Chip L2 Cache Implementation* on page 20. The BPU also contains a 64-entry BTIC that provides immediate access to an instruction pair for taken branches. For more information, see *Section 1.2.2.2 Branch Processing Unit* on page 27.

1.7 Exception Model

The following sections describe the PowerPC exception model and the Espresso implementation. A detailed description of the Espresso exception model is provided in *Section 4 Exceptions* on page 133.

1.7.1 PowerPC Exception Model

The PowerPC exception mechanism allows the processor to interrupt the instruction flow to handle certain situations caused by external signals, errors, or unusual conditions arising from the instruction execution. When exceptions occur, information about the state of the processor is saved to certain registers, and the processor begins execution at an address (exception vector) predetermined for each exception. System software must save the processor state before servicing the exception. Exception processing proceeds in supervisor mode.

Although multiple exception conditions can map to a single exception vector, a more specific condition can be determined by examining a register associated with the exception—for example the MSR, DSISR, and FPSCR contain status bits that further identify the exception condition. Additionally, some exception conditions can be explicitly enabled or disabled by software.

The PowerPC Architecture requires that exceptions be handled in specific priority and program order; therefore, although a particular implementation might recognize exception conditions out of order, they are handled in program order. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that are undispatched, are required to complete before the exception is taken. Any exceptions that those instructions cause must also be handled first. Likewise, asynchronous, precise exceptions are recognized when they occur but are not handled until the instructions currently in the completion queue successfully retire or generate an exception, and the completion queue is emptied.

Unless a catastrophic condition causes a system reset or machine check exception, only one exception is handled at a time. For example, if one instruction encounters multiple exception conditions, those conditions are handled sequentially in priority order. After the exception handler completes, the instruction processing continues until the next exception condition is encountered. Recognizing and handling exception conditions sequentially guarantees system integrity.

When an exception is taken, information about the processor state before the exception was taken is saved in SRR0 and SRR1. Exception handlers must save the information stored in SRR0 and SRR1 early to prevent the program state from being lost due to a system reset and machine check exception or due to an instruction-caused exception in the exception handler, and before re-enabling external interrupts. The exception handler must also save and restore any GPR registers used by the handler.

The PowerPC Architecture supports four types of exceptions:

- Synchronous, precise. These are caused by instructions. All instruction-caused exceptions are handled precisely; that is, the machine state at the time the exception occurs is known and can be completely restored. This means that (excluding the trap and system call exceptions) the address of the instruction at fault is provided to the exception handler and that neither the faulting instruction or subsequent instructions in the code stream complete execution before the exception is taken. Once the exception is processed, execution resumes at the address of the faulting instruction (or at an alternate address provided by the exception handler). When an exception is taken due to a trap or system call instruction, execution resumes at an address provided by the handler.
- Synchronous, imprecise. The PowerPC Architecture defines two imprecise floating-point exception modes: recoverable and nonrecoverable. Even though the Espresso core provides a means to enable the

imprecise modes, it implements these modes identically to the precise mode (that is, enabled floating-point exceptions are always precise).

- Asynchronous, maskable. The PowerPC Architecture defines external and decrementer interrupts as maskable, asynchronous exceptions. When these exceptions occur, their handling is postponed until the next instruction, and any exceptions associated with that instruction, completes execution. If no instructions are in the execution units, the exception is taken immediately upon determination of the correct restart address (for loading SRR0). As shown in *Table 1-5*, the Espresso core implements additional asynchronous, maskable exceptions.
- Asynchronous, nonmaskable. There are two nonmaskable asynchronous exceptions: system reset and the machine check exception. These exceptions might not be recoverable, or might provide a limited degree of recoverability. Exceptions report recoverability through the MSR[RI] bit.

1.7.2 Espresso Microprocessor Exception Implementation

The Espresso exception classes are shown in *Table 1-5*. Although exceptions have other characteristics, such as priority and recoverability, *Table 1-5* describes categories of exceptions Espresso handles uniquely. *Table 1-5* includes no synchronous imprecise exceptions; although the PowerPC Architecture supports imprecise handling of floating-point exceptions, Espresso implements these exception modes precisely.

Table 1-5. Espresso Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Type
Asynchronous, nonmaskable	Imprecise	Machine check, system reset.
Asynchronous, maskable	Precise	External, decrementer, system management, and performance monitor interrupts.
Synchronous	Precise	Instruction-caused exceptions.

Table 1-6 lists the Espresso exceptions and the conditions that cause them. Exceptions specific to the Espresso are identified.

Table 1-6. Exceptions and Conditions (Page 1 of 2)

Exception Type	Vector Offset (hexadecimal)	Causing Conditions
Reserved	00000	—
System reset	00100	Assertion of either $\overline{\text{HRESET}}$ or $\overline{\text{SRESET_Cx}}$ or at power-on reset.
Machine check	00200	Assertion of $\overline{\text{MCP_Cx}}$, DMA queue overflow, DMA look-up misses locked cache, or dcbz_I cache hit. MSR[ME] must be set.
DSI	00300	As specified in the PowerPC Architecture. For TLB misses on load, store, or cache operations, a DSI exception occurs if a page fault occurs.
ISI	00400	As defined by the PowerPC Architecture.
External interrupt	00500	MSR[EE] = '1' and $\overline{\text{INT_Cx}}$ is asserted.
Alignment	00600	A floating-point load/store, stmw , stwcx , lmw , lwarx , eciw , or ecow instruction operand is not word-aligned. A multiple/string load/store operation is attempted in little-endian mode. The operand of dcbz or of dcbz_I is in memory that is write-through-required or caching-inhibited, or the cache is disabled.
1. Espresso-specific.		

Table 1-6. Exceptions and Conditions(Continued) (Page 2 of 2)

Exception Type	Vector Offset (hexadecimal)	Causing Conditions
Program	00700	As defined by the PowerPC Architecture.
Floating-point unavailable	00800	As defined by the PowerPC Architecture.
Decrementer	00900	As defined by the PowerPC Architecture, when the most significant bit of the DEC Register changes from '0' to '1' and MSR[EE] = '1'.
Reserved	00A00 - 00BFF	—
System call	00C00	Execution of the system call (sc) instruction.
Trace	00D00	MSR[SE] = '1' or a branch instruction completes and MSR[BE] = '1'. Unlike the architecture definition, isync does not cause a trace exception
Reserved	00E00	The Espresso core does not generate an exception to this vector. Other PowerPC processors can use this vector for floating-point assist exceptions.
Reserved	00E10 - 00EFF	—
Performance monitor ¹	00F00	The limit specified in a PMC Register is reached and MMCR0[ENINT] = '1'.
Instruction address breakpoint ¹	01300	IABR[0:29] matches EA[0:29] of the next instruction to complete, ABR[TE] matches MSR[IR], and ABR[BE] = '1'.
Reserved	01400 - 02FFF	—

1. Espresso-specific.

1.8 Memory Management

The following sections describe the memory management features of the PowerPC Architecture and the Espresso implementation. A detailed description of the Espresso MMU implementation is provided in *Section 5 Memory Management* on page 151.

1.8.1 PowerPC Memory Management Model

The primary functions of the MMU are to translate logical (effective) addresses to physical addresses for memory accesses and to provide access protection on blocks and pages of memory. There are two types of accesses generated by the Espresso core that require address translation: instruction fetches and data accesses to memory generated by load, store, and cache control instructions.

The PowerPC Architecture defines different resources for 32-bit and 64-bit processors; the Espresso core implements the 32-bit memory management model. The memory-management unit provides two types of memory access models: a block address translation (BAT) model and a virtual address model. The BAT block sizes range from 128 KB to 256 MB and are selectable from high-order effective address bits and have priority over the virtual model. The virtual model employs a 52-bit virtual address space made up of a 24-bit segment address space and a 28-bit effective address space. The virtual model uses a demand paging method with a 4 KB or 128 KB page size. In both models, address translation is done completely by hardware, in parallel with cache accesses, with no additional cycles incurred.

The Espresso MMU also provides independent 8-entry BAT arrays for instructions and data that maintain address translations for blocks of memory. These entries define blocks that can vary from 128 KB to 256 MB. The BAT arrays are maintained by system software. Instructions and data share the same virtual address model, but can operate in separate segment spaces.

The Espresso MMU and exception model support demand-paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand-paged implies that individual pages for data and instructions are loaded into physical memory from the system disk only when they are required by an executing program. Infrequently used pages in memory are returned to disk or discarded if they have not been modified.

The hashed page table is a fixed-sized data structure (the size is determined by the amount of physical memory available to the system) that contains 8-byte entries (PTEs) that define the mapping between virtual pages and physical pages. The page table size is a power of 2, and is boundary aligned in memory based on the size of the table. The page table contains a number of page table entry groups (PTEGs). A PTEG contains eight page table entries (PTEs) of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search operations. A given page translation can be found in one of two possible PTEGs. The size and location in memory of the page table is defined in the SDR1 Register.

Setting MSR[IR] enables instruction address translations, and MSR[DR] enables data address translations. If the bit is cleared, the respective effective address is used as the physical address.

1.8.2 Espresso Memory Management Implementation

The Espresso core implements separate MMUs for instructions and data. It implements a copy of the segment registers in the instruction MMU; however, read and write accesses (**mfsr** and **mtsr**) are handled through the segment registers implemented as part of the data MMU. The Espresso MMU is described in *Section 1.2.3 Memory Management Units (MMUs)* on page 30.

The R (referenced) bit is set in the PTE in memory during a page table search due to a TLB miss. Updates to the changed (C) bit are treated like TLB misses. Again the page table is searched to find the correct PTE to update when the C bit changes from 0 to 1.

1.9 Instruction Timing

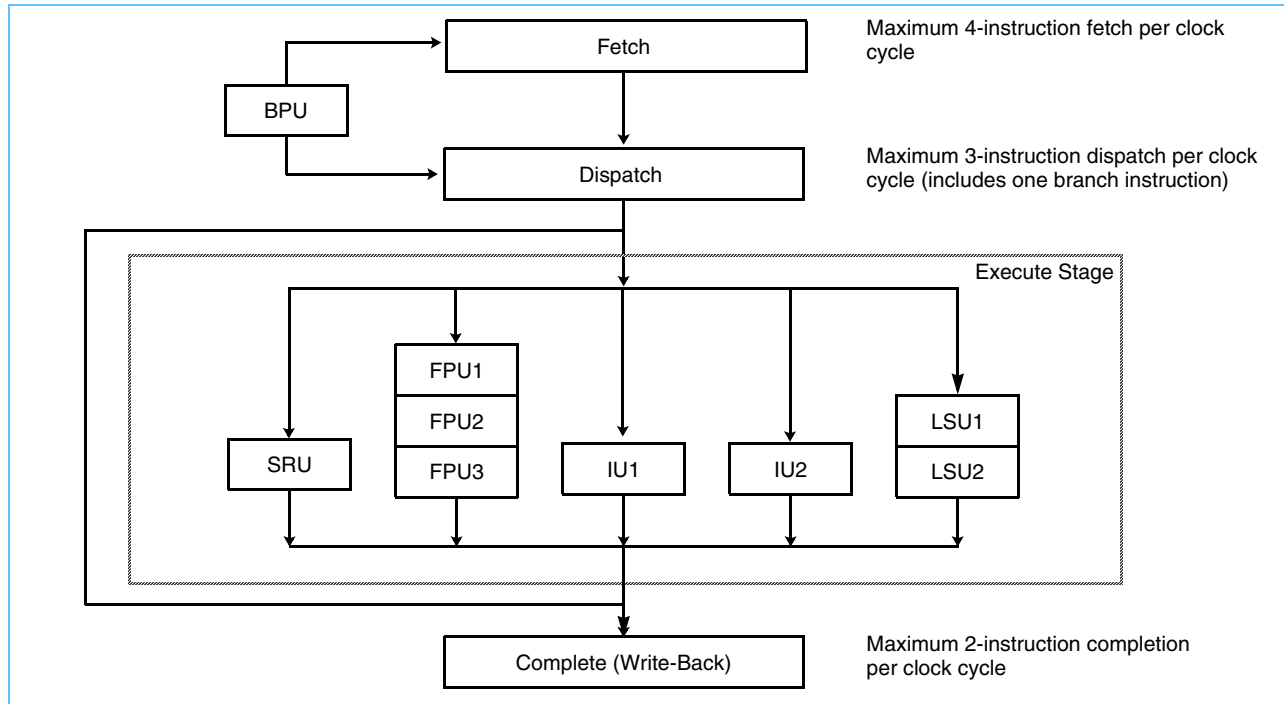
The Espresso microprocessor is a pipelined, superscalar processor. A pipelined processor is one in which instruction processing is divided into discrete stages, allowing work to be done on multiple instructions in each stage. For example, after an instruction completes one stage, it can pass on to the next stage leaving the previous stage available to a subsequent instruction. This improves overall instruction throughput.

A superscalar processor is one that issues multiple independent instructions to separate execution units in a single cycle, allowing multiple instructions to execute in parallel. The Espresso microprocessor has six independent execution units, two for integer instructions, and one each for floating-point instructions, branch instructions, load and store instructions, and system register instructions. Having separate GPRs and FPRs allows integer, floating-point calculations, and load and store operations to occur simultaneously without interference. Additionally, rename buffers are provided to allow operations to post completed results to be used by subsequent instructions without committing them to the architected FPR and GPR register files.

As shown in *Figure 1-4* on page 45, the Espresso common pipeline has four stages through which all instructions must pass: fetch, decode/dispatch, execute, and complete/write back. Instructions flow sequentially through each stage. However, at dispatch a position is made available in the completion queue at the same time it enters the execution stage. This simplifies the completion operation when instructions are retired in program order. Both the load/store and floating-point units have multiple stages to execute their instructions. An instruction occupies only one stage at a time in all execution units. At each stage, an instruction can

proceed without delay or can stall. Stalls are caused by the requirement of additional processing or other events. For example, divide instructions require multiple cycles to complete the operation, load and store instructions can stall waiting for address translation (TLB reload, page fault, and so forth).

Figure 1-4. Pipeline Diagram



Note: Figure 1-4 does not show features such as reservation stations and rename buffers that reduce stalls and improve instruction throughput.

The Espresso instruction pipeline has four major pipeline stages: fetch, dispatch, execute, and complete are described as follows.

- The fetch pipeline stage primarily involves fetching instructions from the memory system and keeping the instruction queue full. The BPU decodes branches after they are fetched and removes (folds out) those that do not update CTR or LR from the instruction stream. If the branch is taken or predicted as taken, the fetch unit is informed of the new address and fetching resumes along the taken patch. For branches not taken or predicted as not taken, sequential fetching continues.
- The dispatch unit is responsible for taking instructions from the bottom two locations of the instruction queue and delivering them to an execution unit for further processing. Dispatch is responsible for decoding the instructions and determining which instructions can be dispatched. To qualify for dispatch, a reservation station, a rename buffer, and a position in the completion queue all must be available. A branch instruction can be processed by the BPU on the same clock cycle for a maximum of three instructions dispatched per cycle.
- The dispatch stage accesses operands, assigns a rename buffer for an operand that updates an architected register (GPR, FPR, CR, and so forth) and delivers the instruction to the reservation registers of the respective execution units. If a source operand is not available (a previous instruction is updating the item using a rename buffer), dispatch provides a tag that indicates which rename buffer supplies the operand when it becomes available. At the end of the dispatch stage, the instructions are removed from

the instructions queue, latched into reservation stations at the appropriate execution unit, and assigned positions in the completion buffers in sequential program order.

- The execution units process instructions from their reservation stations using the operands provided from dispatch and notify the completion stage when the instruction has finished execution. With the exception of multiply and divide, integer instructions complete execution in a single cycle.
- The FPU has three stages for processing floating-point arithmetic. The FPU stages are multiply, add, and normalize. All single-precision arithmetic (add, subtract, multiply, and multiply/add) instructions are processed without stalls at each stage. They have a 1-cycle throughput and a 3-cycle latency. Three different arithmetic instructions can be in execution at one time with one instruction completing execution each cycle. A double-precision arithmetic multiply instruction requires two cycles in the multiply stage, one cycle in the add stage, and one cycle in normalize yielding a 2-cycle throughput and a 4-cycle latency. All divide instructions require multiple cycles in the first stage for processing.
- The load/store unit has two reservation registers and two pipeline stages. The first stage is for effective address calculation, and the second stage is for MMU translation and accessing the L1 data cache. Load instructions have a 1-cycle throughput and a 2-cycle latency.
- In the case of an internal exception, the execution unit reports the exception to the completion pipeline stage and (except for the FPU) discontinues instruction execution until the exception is handled. The exception is not signaled until it is determined that all previous instructions have completed to a point where they do not signal an exception.
- The completion unit retires instructions from the bottom two positions of the completion queue in program order. This maintains the correct architectural machine state and transfers execution results from the rename buffers to the GPRs and FPRs (and CTR and LR, for some instructions) as instructions are retired. If completion logic detects an instruction causing an exception, all following instructions are cancelled, their execution results in rename buffers are discarded, and instructions are fetched from the appropriate exception vector.

Because the PowerPC Architecture can be applied to such a wide variety of implementations, instruction timing varies among PowerPC processors.

For a detailed discussion of instruction timing with examples and a table of latencies for each execution unit, see *Section 6 Instruction Timing* on page 183.

1.10 Performance Monitor

The Espresso microprocessor incorporates a performance monitor facility that system designers can use to help bring up, debug, and optimize software performance. The performance monitor counts events during the execution of code that relates to dispatch, execution, completion, and memory accesses.

The performance monitor incorporates several registers that can be read and written to by supervisor-level software. User-level versions of these registers provide read-only access for user-level applications. These registers are described in *Section 1.4 PowerPC Registers and Programming Model* on page 35. Performance monitor control registers, MMCR0 or MMCR1, can be used to specify which events are to be counted and the conditions for which a performance monitoring interrupt is taken. Additionally, the Sampled Instruction Address Register, SIA (USIA), holds the address of the first instruction to complete after the counter overflowed.

Attempting to write to a user-read-only performance monitor register causes a program exception, regardless of the MSR[PR] setting.



When a performance monitoring interrupt occurs, program execution continues from vector offset x'00F00'.

Section 9 Performance Monitor on page 243 describes the operation of the performance monitor diagnostic tool incorporated in the Espresso microprocessor.



2. Programming Model

This section describes the Espresso programming model, emphasizing those features specific to the Espresso processor and summarizing those that are common to PowerPC processors. It consists of three major sections, which describe the following:

- Registers implemented in Espresso
- Operand conventions
- Espresso instruction set

For detailed information about architecture-defined features, see the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.1 Espresso Processor Register Set

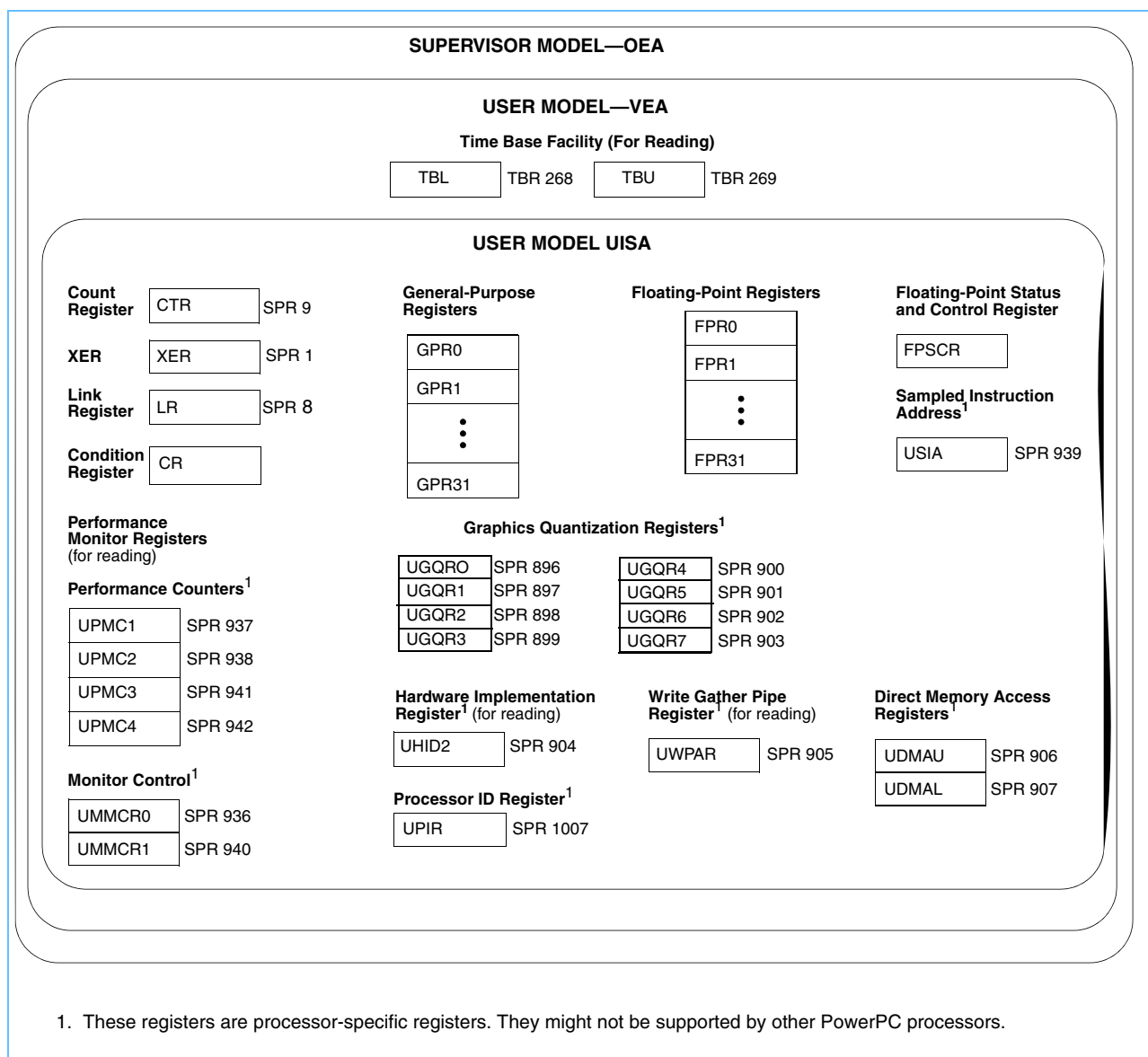
This section includes an overview of the registers defined by the PowerPC Architecture, highlighting differences in how these registers are implemented in Espresso, and a detailed description of the Espresso-specific registers. Full descriptions of the architecture-defined register set are provided in the PowerPC Register Set section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Registers are defined at all three levels of the PowerPC Architecture—user instruction set architecture (UIA), virtual environment architecture (VEA), and operating environment architecture (OEA). The PowerPC Architecture defines register-to-register operations for most computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The 3-register instruction format allows specification of a target register distinct from the two source registers, thus preserving the original data for use by other instructions and reducing the number of instructions required for certain operations. Data is transferred between memory and registers with explicit load and store instructions only.

2.1.1 Register Set

The registers implemented in Espresso are shown in *Figure 2-1 Programming Model (User Model—VEA and UIA)—Espresso Microprocessor Registers* on page 50. The number to the right of the special-purpose registers (SPRs) indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the integer exception register [XER] is SPR 1). These registers can be accessed using the **mtspr** and **mfspr** instructions.

Figure 2-1. Programming Model (User Model—VEA and UISA)—Espresso Microprocessor Registers



The PowerPC UISA registers are user-level. General-purpose registers (GPRs) and floating-point registers (FPRs) are accessed through instruction operands. Access to registers can be explicit (by using instructions for that purpose such as Move to Special-Purpose Register [**mtspr**] and Move from Special-Purpose Register [**mfspr**] instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

Implementation Note: Espresso fully decodes the SPR field of the instruction. If the SPR specified is undefined, an illegal instruction program exception occurs. The PowerPC user-level registers are described as follows:

- **User-level registers (UISA)**—The user-level registers can be accessed by all software with either user or supervisor privileges. They include the following:

- General-purpose registers (GPRs). The 32 GPRs (GPR0 - GPR31) serve as data source or destination registers for integer instructions and provide data for generating addresses. See the General Purpose Registers (GPRs) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
- Floating-point registers (FPRs). The 32 FPRs (FPR0 - FPR31) serve as the data source or destination for all floating-point instructions. See the Floating-Point Registers (FPRs) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Implementation Note: The 32 floating-point registers must be initialized after POR by loading any value from memory, using an **lfd** instruction. This puts the internal representation of each FPR into a valid state. It is necessary to ensure that an FPR is in a valid state before attempting to store the value to memory using a **stfd** instruction.

- Condition Register (CR). The 32-bit CR consists of eight 4-bit fields, CR0 - CR7, that reflect results of certain arithmetic operations and provide a mechanism for testing and branching. See the Condition Register (CR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- Floating-Point Status and Control Register (FPSCR). The FPSCR contains all floating-point exception signal bits, exception summary bits, exception enable bits, and rounding control bits needed for compliance with the IEEE 754 standard. See the Floating-Point Status and Control Register (FPSCR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

The remaining user-level registers are SPRs. Note that the PowerPC Architecture provides a separate mechanism for accessing SPRs (the **mtspr** and **mfspr** instructions). These instructions are commonly used to explicitly access certain registers, while other SPRs can be more typically accessed as the side effect of executing other instructions.

- Integer Exception Register (XER). The XER indicates overflow and carries for integer operations. See the XER Register (XER) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Implementation Note: To allow emulation of the **lscbx** instruction defined by the Power Architecture, XER[16:23] is implemented so that it can be read with **mfspr**[XER] and written with **mtxer**[XER] instructions.

- Link Register (LR). The LR provides the branch target address for the Branch Conditional to Link Register (**bclrx**) instruction and can be used to hold the logical address of the instruction that follows a branch and link instruction, typically used for linking to subroutines. See the Link Register (LR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- Count Register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctrx**) instruction. See the Count Register (CTR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- Processor ID Register (PIR). The PIR is defined in the OEA level of the PowerPC Architecture. However, in Espresso, the UPIR is a user-level register, whose low-order 2 bits can be used by software to identify on which of the three cores (0, 1, or 2) the software is currently running.

In addition to these registers defined in the UISA, Espresso provides user-mode access to several registers that are defined in the OEA. These include performance monitor registers, quantization registers, DMA registers, and the HID2 and WPAR registers. These registers are described along with other supervisor-level registers in this section.

- **User-level registers (VEA)**—The PowerPC VEA defines the time base facility (TB), which consists of two 32-bit registers: Time Base Upper (TBU) and Time Base Lower (TBL). The time base registers can be written to only by supervisor-level instructions but can be read by both user-level and supervisor-level software. For more information, see the PowerPC VEA Register Set—Time Base section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In the Espresso chip, one set of time base registers are implemented. These are shared among the three cores.

- **Supervisor-level registers (OEA)**—The OEA defines the registers that an operating system uses for memory management, configuration, exception handling, and other operating system functions. The OEA defines the following supervisor-level registers for 32-bit implementations:
 - Configuration registers
 - Machine State Register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and Return from Exception (**rfi**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. When an exception is taken, the contents of the MSR are saved to the Machine Status Save/Restore Register 1 (SRR1), which is described below. See the Machine State Register (MSR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

In addition to the synchronization requirements described in the Synchronization Requirements for Special Registers and for Lookaside Buffers section of the the *PowerPC Microprocessor Family: The Programming Environments* manual, an **mtmsr** instruction that modifies MSR[IR] must be followed by a context-synchronizing instruction, such as an **isync**, in all cases.

Implementation Note—Table 2-1 describes the MSR bits that Espresso implements that are not required by the PowerPC Architecture.

Table 2-1. Additional MSR Bits

Bits	Field Name	Description
13	POW	Power management enable. Optional in the PowerPC Architecture.
		0 Power management is disabled. 1 Power management is enabled. The processor can enter a power-saving mode when additional conditions are present. The mode chosen is determined by the DOZE, NAP, and SLEEP bits in the Hardware Implementation-Dependent Register 0 (HID0).
29	PM	Performance monitor marked mode. This bit is specific to Espresso, and is defined as reserved by the PowerPC Architecture. See <i>Section 9 Performance Monitor</i> on page 243.
		0 Process is not a marked process. 1 Process is a marked process.

Note: Setting MSR[EE] masks not only the architecture-defined external interrupt and decremented exceptions but also the Espresso-specific system management and performance monitor exceptions.

- Memory management registers
 - Block Address Translation (BAT) Registers. The PowerPC OEA includes an array of block address translation registers that can be used to specify four blocks of instruction space and four blocks of data space. The BAT registers are implemented in pairs: four pairs of instruction BATs (IBAT0U - IBAT3U and IBAT0L - IBAT3L) and four pairs of data BATs (DBAT0U - DBAT3U and DBAT0L - DBAT3L). The Espresso core supports an enhanced BAT facility that allows specification of eight blocks of instruction space and eight blocks of data space. In this mode, the eight instruction BAT register pairs are IBAT0U - IBAT7U and IBAT0 - BAT7L, and the eight data BAT

register pairs are DBAT0U - DBAT7U and DBAT0L - DBAT7L. For more information, see the BAT Registers section in the *PowerPC Microprocessor Family: The Programming Environments* manual. Because BAT upper and lower words are loaded separately, software must ensure that BAT translations are correct during the time that both BAT entries are being loaded.

Espresso implements the G bit in the IBAT registers; however, attempting to execute code from an IBAT area with G = '1' causes an ISI exception. This complies with the revision of the architecture described in the *PowerPC Microprocessor Family: The Programming Environments* manual.

- Storage Description Register 1 (SDR1). The SDR1 Register specifies the page table base address used in virtual-to-physical address translation. See the SDR1 section in the *PowerPC Microprocessor Family: The Programming Environments* manual."
- Segment Registers (SR). The PowerPC OEA defines sixteen 32-bit segment registers (SR0 - SR15). Note that the SRs are implemented on 32-bit implementations only. The fields in the segment register are interpreted differently depending on the value of bit 0. See the Segment Registers section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Note: Espresso implements separate memory management units (MMUs) for instruction and data. It associates the architecture-defined SRs with the data MMU (DMMU). It reflects the values of the SRs in separate, so-called shadow segment registers in the instruction MMU (IMMU).

- Exception-handling registers
 - Data Address Register (DAR). After a DSI or an alignment exception, DAR is set to the effective address (EA) generated by the instruction causing the fault. See the Data Address Register (DAR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
 - Software Use SPRs (SPRG0 - SPRG3). The SPRG0 - SPRG3 Registers are provided for operating system use. See the SPRG0 - SPRG3 section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
 - Data Storage Interrupt Status Register (DSISR). The DSISR Register defines the cause of DSI and alignment exceptions. See the DSISR section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
 - Machine Status Save/Restore Register 0 (SRR0). The SRR0 Register is used to save the address of the instruction at which execution continues when **rfi** executes at the end of an exception handler routine. See the Machine Status Save/Restore Register 0 (SRR0) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
 - Machine Status Save/Restore Register 1 (SRR1). The SRR1 Register is used to save machine status on exceptions and to restore machine status when **rfi** executes. See the Machine Status Save/Restore Register 1 (SRR1) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Implementation Note: When a machine check exception occurs, Espresso sets one or more error bits in the SRR1. *Table 2-2* describes the SRR1 bits Espresso implements that are not required by the PowerPC Architecture.

Table 2-2. Additional SRR1 Bits

Bit	Field Name	Description
10	DMA	Set by a dcbz_I or DMA error.
12	MCPIN	Set by the assertion of MCP .

- Miscellaneous registers

- Time Base (TB). The TB is a 64-bit structure provided for maintaining the time of day and operating interval timers. The TB consists of two 32-bit registers: Time Base Upper (TBU) and Time Base Lower (TBL). The Time Base Registers can be written to only by supervisor-level software, but can be read by both user-level and supervisor-level software. See the time base facility (TB)—OEA section of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Note: The two time base registers on the Espresso microprocessor, TBU and TBL, are shared among all three cores on the chip.

- Decrementer Register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay; the frequency is a subdivision of the processor clock. See the Decrementer Register (DEC) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Implementation Note: In the Espresso implementation, the Decrementer Register is decremented and the time base is incremented at a speed that is one-fourth the speed of the bus clock.

- Data Address Breakpoint Register (DABR). This optional register is used to cause a breakpoint exception if a specified data address is encountered. See the Data Address Breakpoint Register (DABR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual.
- External Access Register (EAR). This optional register is used in conjunction with the **eciwx** and **ecowx** instructions. Note that the EAR Register and the **eciwx** and **ecowx** instructions are optional in the PowerPC Architecture and might not be supported in all PowerPC processors that implement the OEA. See the External Access Register (EAR) section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.
- Espresso-specific registers—The PowerPC Architecture allows implementation-specific SPRs. The Espresso implementation incorporates the following SPRs. Note that in Espresso these registers are all supervisor-level registers.
 - Direct Memory Access (DMA) Registers. UDMAU and UDMAL provide user-level access to the DMA facility.
 - Graphics Quantization Registers (GQRs). This array of eight registers is used to specify the conversion parameters used by the paired single quantized load and store instructions. UGQR0 - UGQR7 provide user-level access to the quantization registers.
 - Performance Monitor Registers. The following registers are used to define and count events for use by the performance monitor:
 - Performance Monitor Counter Registers (PMC1 - PMC4). These registers are used to record the number of times a certain event has occurred. UPMC1 - UPMC4 provide user-level read access to these registers.

- Monitor Mode Control Registers (MMCR0 - MMCR1). These registers are used to enable various performance monitor interrupt functions. UMMCR0 - UMMCR1 provide user-level read access to these registers.
- Sampled Instruction Address Register (SIA). This register contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. USIA provides user-level read access to the SIA.

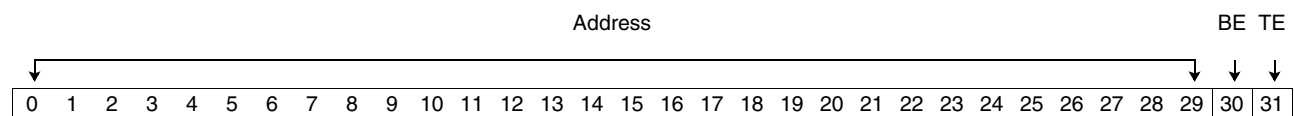
Note: While it is not guaranteed that the implementation of the Espresso-specific registers is consistent among PowerPC processors, other processors can implement similar or identical registers.

2.1.2 Espresso-Specific Registers

This section describes registers that are defined for Espresso but are not included in the PowerPC Architecture.

2.1.2.1 Instruction Address Breakpoint Register (IABR)

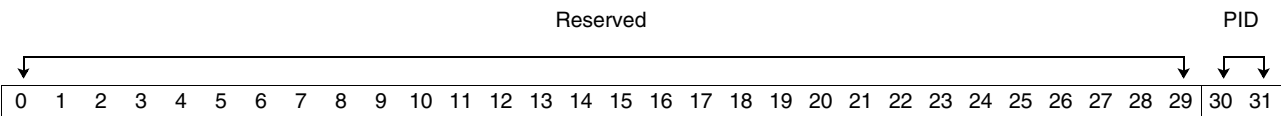
The Instruction Address Breakpoint Register supports the instruction address breakpoint exception. When this exception is enabled, instruction fetch addresses are compared with an effective address stored in the IABR. If the word specified in the IABR is fetched, the instruction breakpoint handler is invoked. The instruction that triggers the breakpoint does not execute before the handler is invoked. For more information, see *Section 4.5.13 Instruction Address Breakpoint Exception (x'01300')* on page 150. The IABR can be accessed with **mtspr** and **mfspr** using the SPR 1010.



Bits	Field Name	Description
0:29	Address	Word address to be compared.
30	BE	Breakpoint enabled. Setting this bit indicates that breakpoint checking is to be performed.
31	TE	Translation enabled. An IABR match is signaled if this bit matches MSR[IR].

2.1.2.2 Processor ID Register (PIR)

The Processor ID Register that is provided for each core enables software to determine on which core it is running. The PIR is a 32-bit, read-only register, whose low-order 2 bits contain the processor ID. UPIR can be accessed with **mfspr** using SPR 1007. The PIR can also be accessed in supervisor mode with **mfspr** using SPR 1023.



Bits	Field Name	Description
0:29	—	Reserved.
30:31	PID	Processor ID. 00 Identifies the current processor as core 0. 01 Identifies the current processor as core 1. 10 Identifies the current processor as core 2. 11 Reserved.

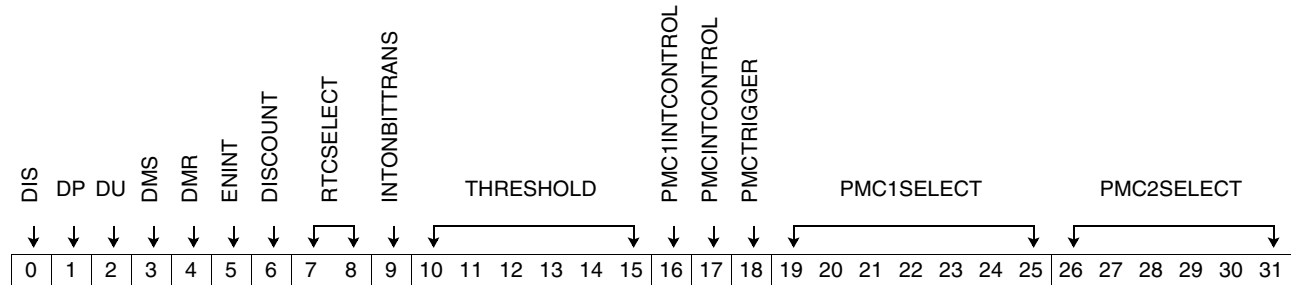
2.1.2.3 Performance Monitor Registers

This section describes the registers used by the performance monitor, which is described in *Section 9 Performance Monitor* on page 243.

Monitor Mode Control Register 0 (MMCR0)

The MMCR0 is a 32-bit SPR provided to specify events to be counted and recorded. The MMCR0 can be accessed only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an **mfspr** instruction to UMMCR0, described in *User Monitor Mode Control Register 1 (UMMCR1)* on page 59.

This register must be cleared at power up. Reading this register does not change its contents.



Bits	Field Name	Description
0	DIS	Disables counting unconditionally. 0 The values of the PMC n counters can be changed by hardware. 1 The values of the PMC n counters cannot be changed by hardware.
1	DP	Disables counting while in supervisor mode. 0 The PMC n counters can be changed by hardware. 1 If the processor is in supervisor mode (MSR[PR] is cleared), the counters are not changed by hardware.
2	DU	Disables counting while in user mode. 0 The PMC n counters can be changed by hardware. 1 If the processor is in user mode (MSR[PR] is set), the PMC n counters are not changed by hardware.
3	DMS	Disables counting while MSR[PM] is set. 0 The PMC n counters can be changed by hardware. 1 If MSR[PM] is set, the PMC n counters are not changed by hardware.
4	DMR	Disables counting while MSR(PM) is zero. 0 The PMC n counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMC n counters are not changed by hardware.
5	ENINT	Enables performance monitor interrupt signaling. 0 Interrupt signaling is disabled. 1 Interrupt signaling is enabled. Cleared by hardware when a performance monitor interrupt is signaled. To reenoble these interrupt signals, software must set this bit after handling the performance monitor interrupt. The initial program load (IPL) ROM code clears this bit before passing control to the operating system.

Bits	Field Name	Description
6	DISCOUNT	Disables counting of PMC_n when a performance monitor interrupt is signaled (that is, ((PMC_nINTCONTROL = '1') & (PMC_n[0] = '1') & (ENINT = '1')) or the occurrence of an enabled time base transition with ((INTONBITTRANS = 1) & (ENINT = 1)). 0 Signaling a performance monitor interrupt does not affect the counting status of PMC_n. 1 The signaling of a performance monitor interrupt prevents changing the PMC1 counter. The PMC_n counter does not change if PMC2COUNTCTL = '0'. Because a time base signal could have occurred along with an enabled counter overflow condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.
7:8	RTCSELECT	64-bit time base, bit selection enable. 00 Pick bit 63 to count. 01 Pick bit 55 to count. 10 Pick bit 51 to count. 11 Pick bit 47 to count.
9	INTONBITTRANS	Cause interrupt signaling on bit transition (identified in RTCSELECT) from off to on. 0 Do not allow an interrupt signal if the chosen bit transitions. 1 Signal interrupt if the chosen bit transitions. Software is responsible for setting and clearing INTONBITTRANS.
10:15	THRESHOLD	Threshold value. Espresso supports all 6 bits, allowing threshold values from 0 - 63. The intent of the THRESHOLD support is to characterize L1 data cache misses.
16	PMC1INTCONTROL	Enables interrupt signaling due to PMC1 counter overflow. 0 Disable PMC1 interrupt signaling due to PMC1 counter overflow. 1 Enable PMC1 interrupt signaling due to PMC1 counter overflow.
17	PMCINTCONTROL	Enable interrupt signaling due to any PMC2 - PMC4 counter overflow. Overrides the setting of DISCOUNT. 0 Disable PMC2 - PMC4 interrupt signaling due to PMC2 - PMC4 counter overflow. 1 Enable PMC2 - PMC4 interrupt signaling due to PMC2 - PMC4 counter overflow.
18	PMC TRIGGER	Can be used to trigger counting of PMC2 - PMC4 after PMC1 has overflowed or after a performance monitor interrupt is signaled. 0 Enable PMC2 - PMC4 counting. 1 Disable PMC2 - PMC4 counting until either PMC1[0] = '1' or a performance monitor interrupt is signaled.
19:25	PMC1 SELECT	PMC1 input selector. 128 events selectable. See <i>Performance Monitor Counter Registers (PMC1 - PMC4)</i> on page 60.
26:31	PMC2 SELECT	PMC2 input selector. 64 events selectable. See <i>Performance Monitor Counter Registers (PMC1 - PMC4)</i> on page 60.

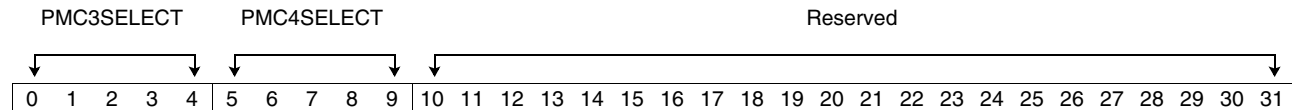
MMCR0 can be accessed with **mtspr** and **mfspr** using SPR 952.

User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, which can be read by user-level software. MMCR0 can be accessed with **mfspir** using SPR 936.

Monitor Mode Control Register 1 (MMCR1)

The MMCR1 functions as an event selector for Performance Monitor Counter Registers 3 and 4 (PMC3 and PMC4).



The corresponding events are described in *Performance Monitor Counter Registers (PMC1 - PMC4)* on page 60.

Bits	Field Name	Description
0:4	PMC3SELECT	PMC3 input selector. 32 events selectable. See <i>Performance Monitor Counter Registers (PMC1 - PMC4)</i> on page 60 for defined selections.
5:9	PMC4SELECT	PMC4 input selector. 32 events selectable. See <i>Performance Monitor Counter Registers (PMC1 - PMC4)</i> on page 60 for defined selections.
10:31	—	Reserved

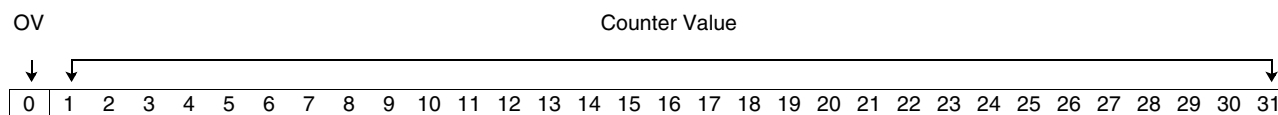
MMCR1 can be accessed with **mtspir** and **mfspir** using SPR 956. User-level software can read the contents of MMCR1 by issuing an **mfspir** instruction to UMMCR1, described in the following section.

User Monitor Mode Control Register 1 (UMMCR1)

The contents of MMCR1 are reflected to UMMCR1, which can be read by user-level software. MMCR1 can be accessed with **mfspir** using SPR 940.

Performance Monitor Counter Registers (PMC1 - PMC4)

The PMC1 - PMC4 Registers are 32-bit counters that can be programmed to generate interrupt signals when they overflow.



Bits	Field Name	Description
0	OV	Overflow. When this bit is set, it indicates that this counter has reached its maximum value.
1:31	Counter Value	Indicates the number of occurrences of the specified event.

Counters are considered to overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value 2147483648 (x'80000000'). However, an interrupt is not signaled unless both `PMCn[INTCONTROL]` and `MMCR0[ENINT]` are also set.

Note: The interrupts can be masked by clearing `MSR[EE]`; the interrupt signal condition can occur with `MSR[EE]` cleared, but the exception is not taken until `EE` is set. Setting `MMCR0[DISCOUNT]` forces counters to stop counting when a counter interrupt occurs.

Software is expected to use `mtspr` to set PMC explicitly to nonoverflow values. If software sets an overflow value, an erroneous exception can occur. For example, if both `PMCn[INTCONTROL]` and `MMCR0[ENINT]` are set and `mtspr` loads an overflow value, an interrupt signal can be generated without any event counting having taken place.

The event to be monitored by PMC1 can be chosen by setting `MMCR0[19:25]`. The event to be monitored by PMC2 can be chosen by setting `MMCR0[26:31]`. The event to be monitored by PMC3 can be chosen by setting `MMCR1[0:4]`. The event to be monitored by PMC4 can be chosen by setting `MMCR1[5:9]`. The selected events are counted beginning when `MMCR0` is set until either `MMCR0` is reset or a performance monitor interrupt is generated.

Table 9-2. PMC1 Events—MMCR0[19:25] Select Encodings on page 248, Table 9-3. PMC2 Events—MMCR0[26:31] Select Encodings on page 249, Table 9-4. PMC3 Events—MMCR1[0:4] Select Encodings on page 250, and Table 9-5. PMC4 Events—MMCR1[5:9] Select Encodings on page 251 list the selectable events and their encodings.

The PMC registers can be accessed with `mtspr` and `mfspr` using the following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958

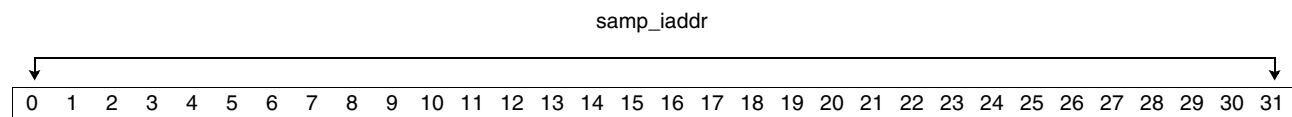
User Performance Monitor Counter Registers (UPMC1 - UPMC4)

The contents of the PMC1 - PMC4 are reflected to UPMC1 - UPMC4, which can be read by user-level software. The UPMC registers can be read with **mfspir** using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942

Sampled Instruction Address Register (SIA)

The SIA Register is a supervisor-level register that contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition.



Bits	Field Name	Description
0:31	samp_iaddr	Sampled instruction address.

If the performance monitor interrupt is triggered by a threshold event, the SIA contains the exact instruction (called the sampled instruction) that caused the counter to overflow.

If the performance monitor interrupt was caused by something besides a threshold event, the SIA contains the address of the last instruction completed during that cycle. SIA can be accessed with the **mtspir** and **mfspir** instructions using SPR 955.

User Sampled Instruction Address Register (USIA)

The contents of SIA are reflected to USIA, which can be read by user-level software. USIA can be accessed with the **mfspir** instructions using SPR 939.

Sampled Data Address Register (SDA) and User Sampled Data Address Register (USDA)

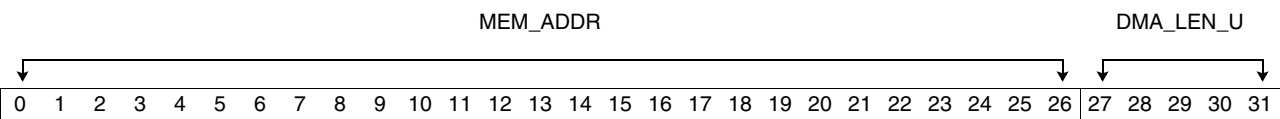
Espresso does not implement the SDA or the user-level, read-only USDA Registers. However, for compatibility with processors that do, those registers can be written to by boot code without causing an exception. SDA is SPR 959; USDA is SPR 943.

2.1.2.4 Direct Memory Access Registers (DMAU and DMAL)

The pair of DMA registers, DMAU and DMAL, is used to specify and issue a DMA command. A DMA command specifies the transfer of a contiguous block of data, up to 4 KB, between the locked cache and external memory. Each DMA command consists of the starting address in locked cache, the starting address in external memory, the length of the transfer in cache lines, and the direction of the transfer.

The DMA facility is enabled using the HID2[LCE] bit. When HID2[LCE] = '0', the **mtspr** and **mfspr** instructions can be used to read and write the DMA registers, but the DMA commands associated with these registers are ignored. In particular, the DMA_T and DMA_F bits in DMAL are always forced to zero in this mode. When HID2[LCE] = '1', an **mtspr** to DMAL with the DMA_F bit = '1' causes the DMA command queue to be flushed; otherwise, an **mtspr** DMAL with the DMA_T bit = '1' causes the DMA command specified in the DMA registers to be added to the DMA command queue.

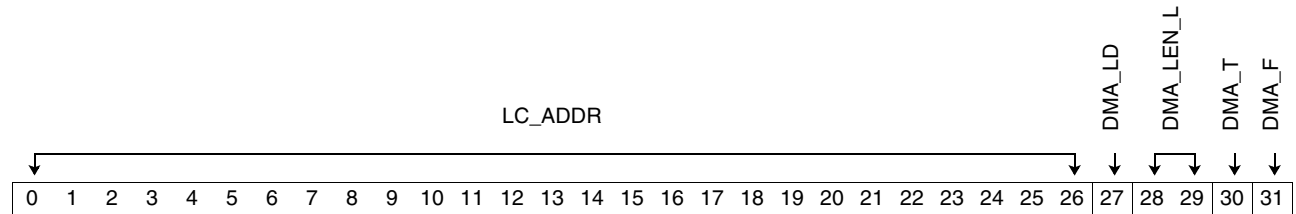
Direct Memory Access Upper (DMAU) Register



Bits	Field Name	Description
0:26	MEM_ADDR	High-order address bits of starting address in external memory of the DMA transfer. The low-order address bits are zero, forcing the starting address to be cache-line aligned.
27:31	DMA_LEN_U	High-order bits of transfer length, in cache lines. The low-order bits are in DMAL.

DMAU can be accessed with **mtspr** and **mfspr** using SPR 922.

Direct Memory Access Lower (DMAL) Register



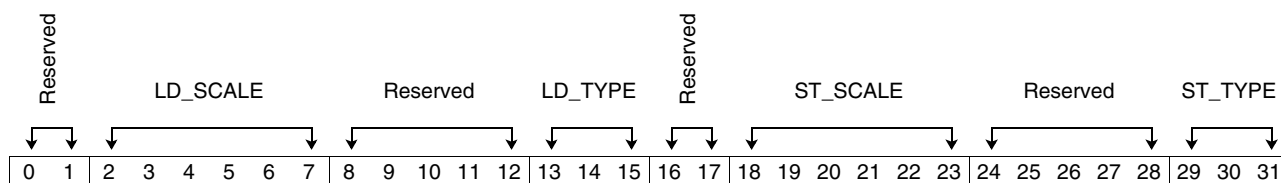
Bits	Field Name	Description
0:26	LC_ADDR	High-order address bits of the starting address in the locked cache of the DMA transfer. The low-order address bits are zero, forcing the starting address to be cache-line aligned.
27	DMA_LD	DMA load command. 0 Store. Transfer from a locked cache to external memory. 1 Load. Transfer from external memory to a locked cache.
28:29	DMA_LEN_L	Low-order bits of transfer length, in cache lines. High-order bits are in the DMAU.
30	DMA_T	Trigger bit. 0 DMA command inactive. 1 mtspr DMAL instruction with this bit active enqueues this DMA command.
31	DMA_F	Flush bit. 0 Normal DMA operation. 1 mtspr DMAL instruction with this bit active flushes the DMA queue.

DMAL can be accessed with **mtspr** and **mfspir** using SPR 923.

UDMAU (SPR 906) and UDMAL (SPR 907) provide user-level access to the DMA facility. The definition of these registers for user-level access is the same as that for supervisor-level access, except that bit 31 of UDMAL is reserved.

2.1.2.5 Graphics Quantization Registers (GQRs)

The eight graphics quantization registers, GQR0 - GQR7, are used to specify the data type and scaling factor used to convert operands in paired single quantized load and store instructions. The specific GQR used for a particular instruction is specified by the 3-bit I field in the instruction.



Bits	Field Name	Description
0:1	—	Reserved.
2:7	LD_SCALE	Scale value used by a load instruction.
8:12	—	Reserved.
13:15	LD_TYPE	Type of operand in memory to be converted by a load instruction. See <i>Table 2-3</i> .
16:17	—	Reserved.
18:23	ST_SCALE	Scale value used by a store instruction.
24:28	—	Reserved.
29:31	ST_TYPE	Type of operand resulting from a conversion by a store instruction. See <i>Table 2-3</i> .

Table 2-3 lists the encoding of the type fields in the GQR for the various quantized data types.

Table 2-3. Quantized Data Types

Code	Type
0	Single-precision floating-point (no conversion)
1:3	Reserved
4	Unsigned 8-bit integer
5	Unsigned 16-bit integer
6	Signed 8-bit integer
7	Signed 16-bit integer

GQR0 - GQR7 can be accessed with **mtspr** and **mfspr** using SPR 912 through 919, respectively.

UGQR0 - UGQR7 (SPRs 896 through 903) provide user-level access to the quantization registers. The definitions of these registers for user-level access is the same as that for supervisor-level access.

2.2 Operand Conventions

This section describes the operand conventions as they are represented in two levels of the PowerPC Architecture: UISA and VEA. Detailed descriptions of conventions used for storing values in registers and memory, accessing PowerPC registers, and representing data in these registers can be found in the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.2.1 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands can be bytes, halfwords, words, or doublewords, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

2.2.2 Alignment and Misaligned Accesses

The operand of a single-register memory access instruction has an alignment boundary equal to its length. An operand address is misaligned if it is not a multiple of its width. Operands for single-register memory access instructions have the characteristics shown in *Table 2-4*. Although not permitted as memory operands, quadwords are shown because quadword alignment is desirable for certain memory operands.

Table 2-4. Memory Operands

Operand	Length	Addr[28:31] if Aligned
Byte	8 bits	xxxx
Halfword	2 bytes	xxx0
Word	4 bytes	xx00
Doubleword	8 bytes	x000
Quadword	16 bytes	0000

Note: An “x” in an address bit position indicates that the bit can be 0 or 1 independent of the state of other bits in the address.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment can affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

Instructions are 32 bits (one word) long and must be word-aligned.

Espresso does not provide hardware support for floating-point memory that is not word-aligned. If a floating-point operand is not aligned, Espresso invokes an alignment exception, and it is left up to software to break up the offending storage access operation appropriately. In addition, some non-doubleword-aligned memory accesses suffer performance degradation as compared to an aligned access of the same type.

In general, floating-point word accesses should always be word-aligned and floating-point doubleword accesses should always be doubleword aligned. Frequent use of misaligned accesses is discouraged, because they can degrade overall performance.

2.2.3 Floating-Point Operand and Execution Models—UIA

The IEEE 754 standard defines conventions for 64- and 32-bit arithmetic. The standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands.

The PowerPC UIA follows these guidelines:

- Double-precision arithmetic instructions can have single-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, conversion from double-precision to single-precision must be done explicitly by software, while conversion from single-precision to double-precision is done implicitly by the processor.

All PowerPC implementations provide the equivalent of the execution models described in the floating-point execution models (UIA) section of the *PowerPC Microprocessor Family: The Programming Environments* manual to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follows conventions described in that section.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is one:

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

Espresso provides hardware support for all single-precision and double-precision floating-point operations for most value representations and all rounding modes. This architecture provides for hardware to implement a floating-point system as defined in ANSI/IEEE standard 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. Detailed information about the floating-point execution model for nonpaired single mode (HID2[PSE] = '0') can be found in the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

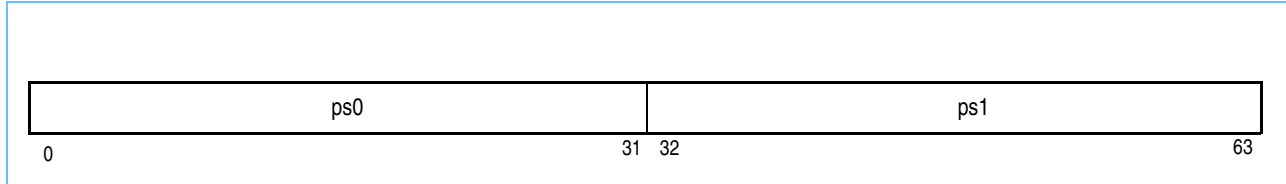
Espresso supports non-IEEE mode whenever FPSCR[29] is set. In this mode, denormalized numbers, NaNs, and some IEEE invalid operations are treated in a non-IEEE conforming manner. This is accomplished by delivering results that approximate the values required by the IEEE standard.

In addition to single-precision and double-precision operands, Espresso supports a third format, called paired single, when HID2[PSE] = '1'. (Note that HID2[PSE] can be changed only when the I-Cache is invalidated and disabled.) Paired single operands are represented in the 64-bit floating-point registers as two 32-bit single-precision floating-point values.

The single-precision floating-point value is referred to in the high-order word as ps0, and in the low-order word as ps1.

Figure 2-2 on page 67 shows the format of an FPR containing a paired single operand.

Figure 2-2. Floating-Point Register Containing a Paired Single Operand



Most of the instructions for manipulating these operands allow both values to be processed in parallel in the execution unit. For example, the paired single multiply-add instruction (**ps_madd**) multiplies ps0 in frA by ps0 in frC, then adds it to ps0 in frB to get a result that is placed in ps0 in frD. Simultaneously, the same operations are applied to the corresponding ps1 values. Note that paired single instructions, including loads, stores, and moves, cause a floating-point unavailable exception if execution is attempted when MSR[FP] = '0'.

Many of the paired single instructions perform an operation comparable to one of the existing double-precision instructions. For example, **fadd** adds double-precision operands from two registers and places the result into a third register. In the corresponding paired single instruction, **ps_add**, two such operations are performed in parallel, one on the ps0 values, and one on the ps1 values. Several other paired single instructions are supported that do not have exact analogies to existing double-precision instructions. See *Section 10 Espresso PowerPC Instruction Set* on page 255 for a detailed description of the paired single instructions.

Most paired single instructions produce a pair of result values. The Floating-Point Status and Control Register (FPSCR) contains a number of status bits that are affected by the floating-point computation. FPSCR bits 15:19 are the result bits. They are determined by the result of the ps0 computation, except for **ps_cmpu1**, **ps_cmpo1**, and **ps_sum1** where the result bits are determined by the result of the ps1 computation. The FPSCR bits that reflect exceptional conditions in the computation are bits 0:14, and 22:23. For paired single instructions that affect any of these bits, either the ps0 or the ps1 computation can set the bit. For the Condition Register (CR), the field specified by **crfD** is affected by the ps0 computation for **ps_cmpo0** and **ps_cmpu0**, and by the ps1 computation for **ps_cmpo1** and **ps_cmpu1**. For all other paired single instructions, when RC = '1', the CR1 field of the CR is set from FPSCR bits 0:3, which can be set by either the ps0 or the ps1 computation.

When in paired single mode (HID2[PSE] = '1'), all the double-precision instructions are still valid, and execute as in nonpaired single mode. In paired single mode, all the single-precision floating-point instructions (**fadds**, **fsubs**, **fmuls**, **fdivs**, **fmadds**, **fmsubs**, **fnmadds**, **fnmsubs**, **fres**, **frsp**) are valid, and operate on the ps0 operand (the double-precision operand, in the case of **frsp**) of the specified registers. The ps1 value in the destination register is duplicated from the ps0 result in such an operation. (See *frspx* on page 336 for an exception about **frsp**.) The load floating-point single instructions (**lfs[u][x]**) load a single-precision floating-point value into the ps0 position of the FPR and duplicate that value in the ps1 position. The store floating-point single instructions (**stfs[u][x]**) store the ps0 value only.

The relationship between the internal format for paired-single operands and that for double-precision floating-point operands is unspecified. It is a programming error to apply double-precision instructions to paired-single operands and vice versa. In particular, loading an operand as a double and then storing it as a paired-single operand does not yield the original value back in memory. This presents a problem when it is required to save the state of FPRs so that they can later be restored, particularly in the case of an interrupt.

The solution to this problem is that the following sequence of store and load instructions, executed when `HID2[PSE] = '1'`, is guaranteed to restore the state of floating-point register **frX** regardless of its format. Assume `GQR0` contains the value 0, indicating that no conversion takes place on paired single quantized loads and stores. Then save each register using the instruction pair:

```
psq_st    frX,0(r1),0,0
stfd      frX,8(r1)
```

and restore each register using the instruction pair:

```
psq_l     frX,0(r1),0,0
lfd       frX,8(r1)
```

When using this restore sequence, a synchronizing operation is required after the **lfd** instruction and before a paired-single instruction that uses the operand in **frX**. This is necessary to ensure that the entire **frX** register is updated before the use of its operand. The intended use of this sequence is in an interrupt handler. Assuming no paired-single instructions are included in the interrupt handler, the **rfi** instruction at the end of the interrupt handler provides this synchronizing function. If this restore sequence is used elsewhere, an **isync** instruction must follow the **lfd** instruction to ensure correct behavior.

Note: Restoration of the ps1 value of a paired single operand is not exact in the following sense. Because the **psq_st** instruction stores a Denorm value as zero, a Denorm on the ps1 side is restored as zero, rather than as the original Denorm value. A Denorm on the ps0 side is correctly saved and restored due to the use of the **stfd** and **lfd** instructions. In addition, there is a subtle data conversion error that can occur when a double-precision Denorm value is stored using the **psq_st** instruction. `HID4[ST0]` can be set to '1' to avoid this error.

Programming Note: Conversion from a double-precision operand to a single-precision operand when `HID2[PSE] = '1'` is accomplished using **frsp**, which takes a double-precision operand as input and produces a single-precision result in ps0 of the destination register. Conversion from a single-precision operand to a double-precision operand, on the other hand, requires a software conversion routine, in general. However, the Espresso processor supports the following performance enhancement to implement this conversion. Any single-precision value in ps0 can be used as the input operand to a double-precision floating-point instruction, including a store.

Note: When `HID2[PSE] = '1'`, the **fctiw** and **fctiwz** instructions give the expected result when used with the **stfiwx** instruction to store the resultant integer. Because these are both classified as double-precision instructions, the integer result is placed in the low-order word of the double-precision operand in the destination FPR. Like other double-precision results, these cannot then be operated on or stored using paired single operations.

Each of the paired single operands or result values behave the same way as single-precision operands or results as shown in the following two tables. *Table 2-5* on page 69 summarizes the conditions and mode behavior for the operands.

Table 2-5. Floating-Point Operand Data Type Behavior

Operand A Data Type	Operand B Data Type	Operand C Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single denormalized Double denormalized	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize all three	Zero all three
Single denormalized Double denormalized	Single denormalized Double denormalized	Normalized or zero	Normalize A and B	Zero A and B
Normalized or zero	Single denormalized Double denormalized	Single denormalized Double denormalized	Normalize B and C	Zero B and C
Single denormalized Double denormalized	Normalized or zero	Single denormalized Double denormalized	Normalize A and C	Zero A and C
Single denormalized Double denormalized	Normalized or zero	Normalized or zero	Normalize A	Zero A
Normalized or zero	Single denormalized Double denormalized	Normalized or zero	Normalize B	Zero B
Normalized or zero	Normalized or zero	Single denormalized Double denormalized	Normalize C	Zero C
Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	Don't care	QNaN ¹	QNaN ¹
Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	Don't care	QNaN ¹	QNaN ¹
Don't care	Don't care	Single QNaN Single SNaN Double QNaN Double SNaN	QNaN ¹	QNaN ¹
Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Single normalized Single infinity Single zero Double normalized Double infinity Double zero	Do the operation	Do the operation
1. Prioritize according to the operand conventions section of the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.				

Table 2-6 summarizes the mode behavior for results.

Table 2-6. Floating-Point Unit Result Data Type Behavior

Precision	Data Type	IEEE Mode (NI = 0)	Non-IEEE Mode (NI = 1)
Single	Denormalized	Return a single-precision denormalized number with trailing zeros.	Return zero. ¹
Single	Normalized, infinity, zero	Return the result.	Return the result. ¹
Single	QNaN, SNaN	Return QNaN.	Return QNaN.
Single	INT	Place an integer into the low word of the FPR.	If (invalid operation) then Place (x'80000000') into FPR[32:63] else Place integer into FPR[32:63].
Double	Denormalized	Return a double-precision denormalized number.	Return zero. ¹
Double	Normalized, infinity, zero	Return the result.	Return the result. ¹
Double	QNaN, SNaN	Return QNaN.	Return QNaN.
Double	INT	Not supported by Espresso.	Not supported by Espresso.

1. The detection of a denorm result occurs before rounding. A result that is a denorm before it is rounded is returned as zero.

2.3 Instruction Set Summary

This section describes the instructions and addressing modes defined for Espresso. These instructions are divided into the following functional categories:

- Integer instructions. These include arithmetic and logical instructions. For more information, see *Section 2.3.4.1 Integer Instructions* on page 77.
- Floating-point instructions. These include floating-point arithmetic instructions (single-precision, double-precision, and paired single), as well as instructions that affect the FPSCR. For more information, see *Section 2.3.4.2 Floating-Point Instructions* on page 80.
- Load and store instructions. These include integer and floating-point (including quantized) load and store instructions. For more information, see *Section 2.3.4.3 Load and Store Instructions* on page 84.
- Flow control instructions. These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see *Section 2.3.4.4 Branch and Flow Control Instructions* on page 96.
- Processor control instructions. These instructions are used for synchronizing memory accesses and managing caches, TLBs, and segment registers. For more information, see *Section 2.3.4.6 Processor Control Instructions (UISA)* on page 98, *Section 2.3.6 Processor Control Instructions (VEA)* on page 104, and *Section 2.3.7.2 Processor Control Instructions (OEA)* on page 109.
- Memory synchronization instructions. These instructions are used for memory synchronizing. For more information, see *Section 2.3.4.7 Memory Synchronization Instructions (UISA)* on page 103 and *Section 2.3.6.1 Memory Synchronization Instructions (VEA)* on page 104.
- Memory control instructions. These instructions provide control of caches, TLBs, and segment registers. For more information, see *Section 2.3.6.2 Memory Control Instructions (VEA)* on page 105 and *Section 2.3.7.3 Memory Control Instructions (OEA)* on page 109.
- External control instructions. These include instructions for use with special input/output devices. For more information, see *Section 2.3.6.3 Optional External Control Instructions* on page 108.

Note: This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions. That information, which is useful for scheduling instructions most effectively, is provided in *Section 6 Instruction Timing* on page 183.

Integer instructions operate on word operands. Floating-point instructions operate on single-precision, double-precision, and paired single floating-point operands. The PowerPC Architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, halfword, and word operand loads and stores between memory and a set of 32 general-purpose registers (GPRs). It provides for word and doubleword operand loads and stores between memory and a set of 32 floating-point registers (FPRs). In addition, the Espresso implementation provides for byte, halfword, word, and doubleword quantized loads and stores between memory and the FPRs.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for some of the frequently-used instructions; see the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete list of simplified mnemonics. Note that the architec-

ture specification refers to simplified mnemonics as extended mnemonics. Programs written to be portable across the various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not described in that document.

2.3.1 Classes of Instructions

The Espresso instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among the PowerPC processors, the assignment of these classifications is not. For example, PowerPC instructions defined for 64-bit implementations are treated as illegal by 32-bit implementations such as Espresso.

The class is determined by examining the primary opcode and the extended opcode, if any. If the opcode, or combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

Instruction encodings that are now illegal might become assigned to instructions in the architecture or might be reserved by being assigned to processor-specific instructions.

2.3.1.1 Definition of Boundedly Undefined

If instructions are encoded with incorrectly set bits in reserved fields, the results on execution can be said to be boundedly undefined. If a user-level program executes the incorrectly coded instruction, the resulting undefined results are bounded in that a spurious change from user to supervisor state is not allowed, and the level of privilege exercised by the program in relation to memory access and other system resources cannot be exceeded. Boundedly-undefined results for a given instruction can vary between implementations, and between execution attempts in the same implementation.

2.3.1.2 Defined Instruction Class

Defined instructions are guaranteed to be supported in all PowerPC implementations, except as stated in the instruction descriptions in *Section 10 Espresso PowerPC Instruction Set* on page 255. Espresso provides hardware support for all instructions defined for 32-bit implementations.

It does not support the optional **fsqrt**, **fsqrts**, and **tlbia** instructions.

A PowerPC processor invokes the illegal instruction error handler (part of the program exception) when unimplemented PowerPC instructions are encountered, so that they can be emulated in software, as required.

Note: The architecture specification refers to exceptions as interrupts.

A defined instruction can have invalid forms. Espresso provides limited support for instructions represented in an invalid form.

2.3.1.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions not defined in the PowerPC Architecture. The following primary opcodes are defined as illegal but might be used in future extensions to the architecture: 1, 5, 6, 9, 22.

Future versions of the PowerPC Architecture might define any of these instructions to perform new functions.

- Instructions defined in the PowerPC Architecture but not implemented in a specific PowerPC implementation. For example, instructions that can be executed on 64-bit PowerPC processors are considered illegal by 32-bit processors such as Espresso.

The following primary opcodes are defined for 64-bit implementations only and are illegal on Espresso: 2, 30, 58, 62.

- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in *Section 2.3.1.4* on page 73 and *Section A.1 Instructions Sorted by Opcode* on page 495. Notice that extended opcodes for instructions defined only for 64-bit implementations are illegal in 32-bit implementations, and vice versa. The following primary opcodes have unused extended opcodes: 4, 17, 19, 31, 59, 63. (Primary opcodes 30 and 62 are illegal for all 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes.)
- An instruction consisting of only zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the system illegal instruction error handler (a program exception).

Note: If only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in *Section 2.3.1.4* on page 73.

Espresso invokes the system illegal instruction error handler (a program exception) when it detects any instruction from this class or any instructions defined only for 64-bit implementations.

See *Section 4.5.6 Program Exception (x'00700')* on page 147 for additional information about illegal and invalid instruction exceptions. Except for an instruction consisting of binary zeros, illegal instructions are available for additions to the PowerPC Architecture.

2.3.1.4 Reserved Instruction Class

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the PowerPC Architecture. Attempting to execute an unimplemented reserved instruction invokes the illegal instruction error handler (a program exception). See *Section 4.5.6 Program Exception (x'00700')* on page 147 for information about illegal and invalid instruction exceptions.

The PowerPC Architecture defines four types of reserved instructions:

- Instructions in the Power Architecture not part of the PowerPC UISA. For details on Power Architecture incompatibilities and how they are handled by PowerPC processors, see the Power Architecture cross-reference section of the *PowerPC Microprocessor Family: The Programming Environments* manual.
- Implementation-specific instructions required for the processor to conform to the PowerPC Architecture (none of these are implemented in Espresso).
- All other implementation-specific instructions.
- Architecturally-allowed extended opcodes.

2.3.1.5 Espresso Implementation-Specific Instructions

The Espresso microprocessor includes extensions to the PowerPC Architecture to enhance the performance of graphics applications. The instructions include a cache control instruction, **dcbz_I**, four quantized load, four quantized store instructions, and 29 paired single floating-point instructions. These instructions are implemented using primary opcodes 4, 56, 57, 60, and 61. See *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229 for a description of the graphics enhancement features and *Section 10 Espresso PowerPC Instruction Set* on page 255 for a detailed description of the instructions.

2.3.2 Addressing Modes

This section provides an overview of conventions for addressing memory and for calculating effective addresses as defined by the PowerPC Architecture for 32-bit implementations. For more detailed information, see the conventions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.2.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a memory access or branch instruction or when it fetches the next sequential instruction.

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

2.3.2.2 Memory Operands

Memory operands can be bytes, halfwords, words, or doublewords, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The PowerPC Architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian. See the byte ordering section of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information about big-endian and little-endian byte ordering.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise, it is misaligned.

For a detailed description about memory operands, see the operand conventions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.2.3 Effective Address Calculation

An effective address is the 32-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit signed twos complement binary arithmetic. A carry from bit 0 and overflow are ignored.

Load and store operations have the following modes of effective address generation:

- $EA = (rAIO) + \text{offset}$ (including offset = '0') (register indirect with immediate index)
- $EA = (rAIO) + rB$ (register indirect with index)

See *Integer Load and Store Address Generation* on page 85 for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate
- Link register indirect
- Count register indirect

2.3.2.4 Synchronization

The synchronization described in this section refers to the state of the processor that is performing the synchronization.

Context Synchronization

The System Call (**sc**) and Return from Interrupt (**rfi**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a change in context. Execution of one of these instructions ensures the following:

- No higher priority exception exists (**sc**).
- All previous instructions have completed to a point where they can no longer cause an exception. If a prior memory access instruction causes direct-store error exceptions, the results are guaranteed to be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following the **sc** or **rfi** instruction execute in the context established by these instructions.

Execution Synchronization

An instruction is execution synchronizing if all previously initiated instructions appear to have completed before the instruction is initiated or, in the case of **sync** and **isync**, before the instruction completes. For example, the Move to Machine State Register (**mtmsr**) instruction is execution synchronizing. It ensures that all preceding instructions have completed execution and cannot cause an exception before the instruction executes, but does not ensure that subsequent instructions execute in the newly established environment.

For example, if the **mtmsr** sets the MSR[PR] bit, unless an **isync** immediately follows the **mtmsr** instruction, a privileged instruction could be executed or privileged access could be performed without causing an exception even though the MSR[PR] bit indicates user mode.

Instruction-Related Exceptions

There are two kinds of exceptions in Espresso: those caused directly by the execution of an instruction and those caused by an asynchronous event (or interrupts). Either can cause components of the system software to be invoked.

Exceptions can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program exception) handler to be invoked. Note that the **dcbz_l** instruction is illegal when HID2[LCE] = '0', the **psq_l**, **psq_lu**, **psq_st**, and **psq_stu** instructions are illegal when HID2[PSQE] = '0' or HID2[PSE] = '0', and all other paired single instructions are illegal when HID2[PSE] = '0'. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program exception) handler to be invoked. Espresso provides the following supervisor-level instructions: **dcbi**, **dcbz_l**, **mfmsr**, **mfmspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbie**, and **tlbsync**. Note that the privilege level of the **mfmspr** and **mtspr** instructions depends on the SPR encoding.
- Any **mtspr**, **mfmspr**, or **mftb** instruction with an invalid SPR (or TBR) field causes an illegal type program exception. Likewise, a program exception is taken if user-level software tries to access a supervisor-level SPR. An **mtspr** instruction executing in supervisor mode (MSR[PR] = '0') with the SPR field specifying HID1 or PVR (read-only registers) executes as a no-op.
- An attempt to access memory that is not available (page fault) causes the ISI or DSI exception handler to be invoked.
- The execution of an **sc** instruction invokes the system call exception handler that permits a program to request the system to perform a service.
- The execution of a trap instruction invokes the program exception trap handler.
- The execution of an instruction that causes a floating-point exception while exceptions are enabled in the MSR invokes the program exception handler.

A detailed description of exception conditions is provided in *Section 4 Exceptions* on page 133.

2.3.3 Instruction Set Overview

This section provides a brief overview of the PowerPC instructions implemented in Espresso and highlights any special information with respect to how Espresso implements a particular instruction. Note that the categories used in this section correspond to those used in the addressing modes and instruction set summary section of the *PowerPC Microprocessor Family: The Programming Environments* manual. These categorizations are somewhat arbitrary and are provided for the convenience of the programmer and do not necessarily reflect the PowerPC Architecture specification.

Note that some instructions have the following optional features:

- CR Update: The dot (.) suffix on the mnemonic enables the update of the CR.
- Overflow option: The **o** suffix indicates that the overflow bit in the XER is enabled.

2.3.4 PowerPC UISA Instructions

The PowerPC UISA includes the base user-level instruction set (excluding a few user-level cache control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

2.3.4.1 Integer Instructions

The integer instructions consist of the following:

- Integer arithmetic instructions
- Integer compare instructions
- Integer logical instructions
- Integer rotate and shift instructions

Integer instructions use the content of the GPRs as source operands and place results into GPRs, into the Integer Exception Register (XER), and into Condition Register (CR) fields.

Integer Arithmetic Instructions

Table 2-7 lists the integer arithmetic instructions for the PowerPC processors.

Table 2-7. Integer Arithmetic Instructions

Name	Mnemonic	Syntax
Add Immediate	addi	rD,rA,SIMM
Add Immediate Shifted	addis	rD,rA,SIMM
Add	add (add. addo addo.)	rD,rA,rB
Subtract From	subf (subf. subfo subfo.)	rD,rA,rB
Add Immediate Carrying	addic	rD,rA,SIMM
Add Immediate Carrying and Record	addic.	rD,rA,SIMM
Subtract from Immediate Carrying	subfic	rD,rA,SIMM
Add Carrying	addc (addc. addco addco.)	rD,rA,rB
Subtract from Carrying	subfc (subfc. subfco subfco.)	rD,rA,rB
Add Extended	adde (adde. addeo addeo.)	rD,rA,rB
Subtract from Extended	subfe (subfe. subfeo subfeo.)	rD,rA,rB
Add to Minus One Extended	addme (addme. addmeo addmeo.)	rD,rA
Subtract from Minus One Extended	subfme (subfme. subfmeo subfmeo.)	rD,rA
Add to Zero Extended	addze (addze. addzeo addzeo.)	rD,rA
Subtract from Zero Extended	subfze (subfze. subfzeo subfzeo.)	rD,rA
Negate	neg (neg. nego nego.)	rD,rA
Multiply Low Immediate	mulli	rD,rA,SIMM
Multiply Low	mullw (mullw. mullwo mullwo.)	rD,rA,rB
Multiply High Word	mulhw (mulhw.)	rD,rA,rB

Table 2-7. Integer Arithmetic Instructions(Continued)

Name	Mnemonic	Syntax
Multiply High Word Unsigned	mulhwu (mulhwu.)	rD,rA,rB
Divide Word	divw (divw. divwo divwo.)	rD,rA,rB
Divide Word Unsigned	divwu divwu. divwuo divwuo.	rD,rA,rB

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**rA**) from the third operand (**rB**). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for examples.

The UISA states that an implementation that executes instructions that set the overflow enable bit (OE) or the carry bit (CA) can either execute these instructions slowly or prevent execution of the subsequent instruction until the operation completes. *Section 6 Instruction Timing* on page 183 describes how Espresso handles CR dependencies. The summary overflow bit (SO) and overflow bit (OV) in the integer exception register are set to reflect an overflow condition of a 32-bit result. This can happen only when OE = '1'.

Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **rA** with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of register **rB**. The comparison is signed for the **cmpi** and **cmp** instructions and unsigned for the **cmpli** and **cmpl** instructions.

Table 2-8 summarizes the integer compare instructions.

Table 2-8. Integer Compare Instructions

Name	Mnemonic	Syntax
Compare Immediate	cmpi	crfD,L,rA,SIMM
Compare	cmp	crfD,L,rA,rB
Compare Logical Immediate	cmpli	crfD,L,rA,UIMM
Compare Logical	cmpl	crfD,L,rA,rB

The **crfD** operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise, the target CR field must be specified in **crfD** using an explicit field number.

For information about simplified mnemonics for the integer compare instructions, see the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Integer Logical Instructions

The logical instructions shown in *Table 2-9* perform bit-parallel operations on the specified operands. Logical instructions with the CR updating enabled (using the dot suffix) and instructions **andi.** and **andis.** set CR field CR0 to characterize the result of the logical operation. Logical instructions do not affect XER[SO], XER[OV], or XER[CA].

See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for simplified mnemonic examples for integer logical operations.

Table 2-9. Integer Logical Instructions

Name	Mnemonic	Syntax	Implementation Notes
AND Immediate	andi.	rA,rS,UIMM	—
AND Immediate Shifted	andis.	rA,rS,UIMM	—
OR Immediate	ori	rA,rS,UIMM	The PowerPC Architecture defines ori r0,r0,0 as the preferred form for the no-op instruction. The dispatcher discards this instruction (except for pending trace or breakpoint exceptions).
OR Immediate Shifted	oris	rA,rS,UIMM	—
XOR Immediate	xori	rA,rS,UIMM	—
XOR Immediate Shifted	xoris	rA,rS,UIMM	—
AND	and (and.)	rA,rS,rB	—
OR	or (or.)	rA,rS,rB	—
XOR	xor (xor.)	rA,rS,rB	—
NAND	nand (nand.)	rA,rS,rB	—
NOR	nor (nor.)	rA,rS,rB	—
Equivalent	eqv (eqv.)	rA,rS,rB	—
AND with Complement	andc (andc.)	rA,rS,rB	—
OR with Complement	orc (orc.)	rA,rS,rB	—
Extend Sign Byte	extsb (extsb.)	rA,rS	—
Extend Sign Halfword	extsh (extsh.)	rA,rS	—
Count Leading Zeros Word	cntlzw (cntlzw.)	rA,rS	—

Integer Rotate Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete list of simplified mnemonics that allow simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1, the associated bit of the rotated data is placed into the target register; if the mask bit is 0, the associated bit in the target register is unchanged) or ANDed with a mask before being placed into the target register.

The integer rotate instructions are summarized in *Table 2-10*.

Table 2-10. Integer Rotate Instructions

Name	Mnemonic	Syntax
Rotate Left Word Immediate then AND with Mask	rlwinm (rlwinm.)	rA,rS,SH,MB,ME
Rotate Left Word then AND with Mask	rlwnm (rlwnm.)	rA,rS,rB,MB,ME
Rotate Left Word Immediate then Mask Insert	rlwimi (rlwimi.)	rA,rS,SH,MB,ME

Integer Shift Instructions

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics are provided to make coding of such shifts simpler and easier to understand.

Multiple-precision shifts can be programmed as shown in the multiple-precision shifts section of the *PowerPC Microprocessor Family: The Programming Environments* manual. The integer shift instructions are summarized in *Table 2-11*.

Table 2-11. Integer Shift Instructions

Name	Mnemonic	Syntax
Shift Left Word	slw (slw.)	rA,rS,rB
Shift Right Word	srw (srw.)	rA,rS,rB
Shift Right Algebraic Word Immediate	srawi (srawi.)	rA,rS,SH
Shift Right Algebraic Word	sraw (sraw.)	rA,rS,rB

2.3.4.2 Floating-Point Instructions

The floating-point instructions include the following:

- Floating-point arithmetic instructions
- Floating-point multiply-add instructions
- Floating-point rounding and conversion instructions
- Floating-point compare instructions
- Floating-point status and control register instructions
- Floating-point move instructions

See *Section 2.3.4.3* on page 84 for information about floating-point loads and stores.

The PowerPC Architecture supports a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard. All floating-point operations conform to the IEEE 754 standard, unless software sets the non-IEEE mode with FPSCR[NI].

Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in *Table 2-12*.

Table 2-12. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax
Floating Add (Double-Precision)	fadd (fadd.)	frD,frA,frB
Floating Add Single	fadds (fadds.)	frD,frA,frB
Floating Subtract (Double-Precision)	fsub (fsub.)	frD,frA,frB
Floating Subtract Single	fsubs (fsubs.)	frD,frA,frB
Floating Multiply (Double-Precision)	fmul (fmul.)	frD,frA,frC
Floating Multiply Single	fmuls (fmuls.)	frD,frA,frC
Floating Divide (Double-Precision)	fdiv (fdiv.)	frD,frA,frB
Floating Divide Single	fdivs (fdivs.)	frD,frA,frB
Floating Reciprocal Estimate Single ¹	fres (fres.)	frD,frB
Floating Reciprocal Square Root Estimate ¹	frsqte (frsqte.)	frD,frB
Floating Select ¹	fsel (fsel.)	frD,frA,frC,frB
Paired Single Add ²	ps_add (ps_add.)	frD,frA,frB
Paired Single Subtract ²	ps_sub (ps_sub.)	frD,frA,frB
Paired Single Multiply ²	ps_mul (ps_mul.)	frD,frA,frC
Paired Single Divide ²	ps_div (ps_div.)	frD,frA,frB
Paired Single Reciprocal Estimate ²	ps_res (ps_res.)	frD,frB
Paired Single Reciprocal Square Root Estimate ²	ps_rsqte (ps_rsqte.)	frD,frB
Paired Single Select ²	ps_sel (ps_sel.)	frD,frA,frC,frB
Paired Single Multiply Scalar High ²	ps_muls0 (ps_muls0.)	frD,frA,frC
Paired Single Multiply Scalar Low ²	ps_muls1 (ps_muls1.)	frD,frA,frC
Paired Single Vector Sum High ²	ps_sum0 (ps_sum0.)	frD,frA,frC,frB
Paired Single Vector Sum Low ²	ps_sum1 (ps_sum1.)	frD,frA,frC,frB

Note:

1. The **fres**, **frsqte** and **fsel** instructions are optional in the PowerPC Architecture.
2. These instructions belong to the Espresso graphics extensions, and are legal only when HID2[PSE] = '1'.

The estimate instructions (**fres**, **frsqte**, **ps_res**, **ps_rsqte**) do not generate an intermediate result. The final result produced by these instructions is independent of the FPSCR[RN] field, and corresponds to the round-to-zero rounding mode

Double-precision arithmetic instructions, except those involving multiplication (**fmul**, **fmadd**, **fmsub**, **fnmadd**, **fnmsub**) execute with the same latency as their single-precision equivalents. For additional details on floating-point performance, see *Section 6 Instruction Timing* on page 183.

Floating-Point Multiply-Add Instructions

These instructions combine multiply and add operations without an intermediate rounding operation. The floating-point multiply-add instructions are summarized in *Table 2-13*.

Table 2-13. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax
Floating Multiply-Add (Double-Precision)	fmadd (fmadd.)	frD,frA,frC,frB
Floating Multiply-Add Single	fmadds (fmadds.)	frD,frA,frC,frB
Floating Multiply-Subtract (Double-Precision)	fmsub (fmsub.)	frD,frA,frC,frB
Floating Multiply-Subtract Single	fmsubs (fmsubs.)	frD,frA,frC,frB
Floating Negative Multiply-Add (Double-Precision)	fnmadd (fnmadd.)	frD,frA,frC,frB
Floating Negative Multiply-Add Single	fnmadds (fnmadds.)	frD,frA,frC,frB
Floating Negative Multiply-Subtract (Double-Precision)	fnmsub (fnmsub.)	frD,frA,frC,frB
Floating Negative Multiply-Subtract Single	fnmsubs (fnmsubs.)	frD,frA,frC,frB
Paired Single Multiply-Add ¹	ps_madd (ps_madd.)	frD,frA,frC,frB
Paired Single Multiply-Subtract ¹	ps_msub (ps_msub.)	frD,frA,frC,frB
Paired Single Negative Multiply-Add ¹	ps_nmadd (ps_nmadd.)	frD,frA,frC,frB
Paired Single Negative Multiply-Subtract ¹	ps_nmsub (ps_nmsub.)	frD,frA,frC,frB
Paired Single Multiply-Add Scalar High ¹	ps_madds0 (ps_madds0.)	frD,frA,frC,frB
Paired Single Multiply-Add Scalar Low ¹	ps_madds1 (ps_madds1.)	frD,frA,frC,frB

1. These instructions are Espresso-specific, and are legal only when HID2[PSE] = '1'.

Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point conversion instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

Examples of uses of these instructions to perform various conversions can be found in the floating-point models section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 2-14. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax
Floating Round to Single	frsp (frsp.)	frD,frB
Floating Convert to Integer Word	fctiw (fctiw.)	frD,frB
Floating Convert to Integer Word with Round toward Zero	fctiwz (fctiwz.)	frD,frB

Floating-Point Comparison Instructions

Floating-point comparison instructions compare the contents of two floating-point registers. The comparison ignores the sign of zero (that is $+0 = -0$).

The floating-point comparison instructions are summarized in *Table 2-15*.

Table 2-15. Floating-Point Comparison Instructions

Name	Mnemonic	Syntax
Floating Compare Unordered	fcmpu	crfD,frA,frB
Floating Compare Ordered	fcmpo	crfD,frA,frB
Paired Single Compare Unordered High ¹	ps_cmpu0	crfD,frA,frB
Paired Single Compare Unordered Low ¹	ps_cmpu1	crfD,frA,frB
Paired Single Compare Ordered High ¹	ps_cmpo0	crfD,frA,frB
Paired Single Compare Ordered Low ¹	ps_cmpo1	crfD,frA,frB

1. These instructions are Espresso-specific and are legal only when $HID2[PSE] = '1'$.

The PowerPC Architecture allows an **fcmpu** or **fcmpo** instruction with the Rc bit set to produce a boundedly-undefined result, which can include an illegal instruction program exception. In Espresso, **crfD** should be treated as undefined

Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed.

The FPSCR instructions are summarized in *Table 2-16*.

Table 2-16. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Syntax
Move from FPSCR	mffs (mffs.)	frD
Move to Condition Register from FPSCR	mcrfs	crfD,crfS
Move to FPSCR Field Immediate	mtfsfi (mtfsfi.)	crfD,IMM
Move to FPSCR Fields	mtfsf (mtfsf.)	FM,frB
Move to FPSCR Bit 0	mtfsb0 (mtfsb0.)	crbD
Move to FPSCR Bit 1	mtfsb1 (mtfsb1.)	crbD

Implementation Note: The PowerPC Architecture states that in some implementations the Move to FPSCR Fields (**mtfsf**) instruction can perform more slowly when only some of the fields are updated as opposed to all of the fields. In the Espresso implementation, there is no degradation of performance.

Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another. The floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1.

Table 2-17 summarizes the floating-point move instructions.

Table 2-17. Floating-Point Move Instructions

Name	Mnemonic	Syntax
Floating Move Register	fmr (fmr.)	frD,frB
Floating Negate	fneg (fneg.)	frD,frB
Floating Absolute Value	fabs (fabs.)	frD,frB
Floating Negative Absolute Value	fnabs (fnabs.)	frD,frB
Paired Single Move Register ¹	ps_mr (ps_mr.)	frD,frB
Paired Single Negate ¹	ps_neg (ps_neg.)	frD,frB
Paired Single Absolute Value ¹	ps_abs (ps_abs.)	frD,frB
Paired Single Negative Absolute Value ¹	ps_nabs (ps_nabs.)	frD,frB
Paired Single Merge High ¹	ps_merge00 (ps_merge00.)	frD,frA,frB
Paired Single Merge Direct ¹	ps_merge01 (ps_merge01.)	frD,frA,frB
Paired Single Merge Swapped ¹	ps_merge10 (ps_merge10.)	frD,frA,frB
Paired Single Merge Low ¹	ps_merge11 (ps_merge11.)	frD,frA,frB

1. These instructions belong to the Espresso graphics extensions, and are legal only when HID2[PSE] = '1'.

2.3.4.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. The load and store instructions consist of the following types of instructions:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions, including quantized loads
- Floating-point store instructions, including quantized stores
- Memory synchronization instructions

Implementation Notes: Espresso provides hardware support for misaligned memory accesses. It performs those accesses within a single cycle if the operand lies within a doubleword boundary. Misaligned memory accesses that cross a doubleword boundary degrade performance.

For string operations, the hardware makes no attempt to combine register values to reduce the number of discrete accesses. Combining stores enhances performance if store gathering is enabled and the accesses meet the criteria described in *Section 6.4.7 Integer Store Gathering* on page 207. Note that the PowerPC Architecture requires load/store multiple instruction accesses to be aligned. At a minimum, additional cache access cycles are required.

Although many unaligned memory accesses are supported in hardware, the frequent use of them is discouraged, because they can compromise the overall performance of the processor.

Accesses that cross a translation boundary can be restarted. That is, a misaligned access that crosses a page boundary is completely restarted if the second portion of the access causes a page fault. This can cause the first access to be repeated.

On Espresso, TLB reloads are done transparently and only a page fault causes a restart.

Self-Modifying Code

When a processor modifies a memory location that can be contained in the instruction cache, software must ensure that memory updates are visible to the instruction fetching mechanism. This can be achieved by the following instruction sequence:

dcbst	! update memory
sync	! wait for update
icbi	! remove (invalidate) copy in instruction cache
sync	! wait for ICBI operation to be globally performed
isync	! remove copy in own instruction buffer

These operations are required because the data cache is a write-back cache. Because instruction fetching bypasses the data cache, changes to items in the data cache might not be reflected in memory until the fetch operations complete.

Special care must be taken to avoid coherency paradoxes in systems that implement unified secondary caches, and designers should carefully follow the guidelines for maintaining cache coherency that are provided in the VEA, and described in the cache model and memory coherency section of the *PowerPC Microprocessor Family: The Programming Environments* manual. Because Espresso does not broadcast the M bit for instruction fetches, external caches are subject to coherency paradoxes.

Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See *Section 2.3.2.3 Effective Address Calculation* on page 75 for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned might suffer performance degradation. See *Section 4.5.5 Alignment Exception (x'00600')* on page 147 for additional information about load and store address alignment exceptions.

Integer Load Instructions

For integer load instructions, the byte, halfword, or word addressed by the effective address (EA) is loaded into **rD**. Many integer load instructions have an update form, in which **rA** is updated with the generated effective address. For these forms, if **rA** $\neq$ 0 and **rA** $\neq$ **rD** (otherwise, invalid), the EA is placed into **rA** and the memory element (byte, halfword, or word) addressed by the EA is loaded into **rD**. Note that the PowerPC Architecture defines load with update instructions with operand **rA** = 0 or **rA** = **rD** as invalid forms.

Table 2-18 summarizes the integer load instructions.

Table 2-18. Integer Load Instructions

Name	Mnemonic	Syntax
Load Byte and Zero	lbz	rD,d(rA)
Load Byte and Zero Indexed	lbzx	rD,rA,rB
Load Byte and Zero with Update	lbzu	rD,d(rA)
Load Byte and Zero with Update Indexed	lbzux	rD,rA,rB
Load Halfword and Zero	lhz	rD,d(rA)
Load Halfword and Zero Indexed	lhzx	rD,rA,rB
Load Halfword and Zero with Update	lhzu	rD,d(rA)
Load Halfword and Zero with Update Indexed	lhzux	rD,rA,rB
Load Halfword Algebraic	lha	rD,d(rA)
Load Halfword Algebraic Indexed	lhax	rD,rA,rB
Load Halfword Algebraic with Update	lhau	rD,d(rA)
Load Halfword Algebraic with Update Indexed	lhaux	rD,rA,rB
Load Word and Zero	lwz	rD,d(rA)
Load Word and Zero Indexed	lwzx	rD,rA,rB
Load Word and Zero with Update	lwzu	rD,d(rA)
Load Word and Zero with Update Indexed	lwzux	rD,rA,rB

Implementation Notes: The following notes describe the Espresso implementation of integer load instructions:

- The PowerPC Architecture cautions programmers that some implementations of the architecture might execute the load half algebraic (**lha**, **lhax**) instructions with greater latency than other types of load instructions. This is not the case for the Espresso implementation; these instructions operate with the same latency as other load instructions.
- The PowerPC Architecture cautions programmers that some implementations of the architecture might run the load/store byte-reverse (**lhbrx**, **lbrx**, **sthbrx**, **stwbrx**) instructions with greater latency than other types of load/store instructions. This is not the case for the Espresso implementation. These instructions operate with the same latency as the other load/store instructions.
- The PowerPC Architecture describes some preferred instruction forms for load and store multiple instructions and integer move assist instructions that might perform better than other forms in some implementations. None of these preferred forms affect instruction performance on the Espresso implementation.
- The PowerPC Architecture defines the **lwarx** and **stwcx** as a way to update memory atomically. In the Espresso implementation, reservations are made on behalf of aligned 32-byte sections of the memory

address space. Executing **lwarx** and **stwcx**. to a page marked write-through does not cause a DSI exception if the W bit is set, but as with other memory accesses, DSI exceptions can result for other reasons such as protection violations or page faults.

- In general, because **stwcx**. always causes an external bus transaction, it has slightly worse performance characteristics than normal store operations.

Integer Store Instructions

For integer store instructions, the contents of **rS** are stored into the byte, halfword, or word in memory addressed by the EA. Many store instructions have an update form, in which **rA** is updated with the EA. For these forms, the following rules apply:

- If **rA** $\neq$ 0, the effective address is placed into **rA**.
- If **rS** = **rA**, the contents of register **rS** are copied to the target memory element, then the generated EA is placed into **rA** (**rS**).

The PowerPC Architecture defines store with update instructions with **rA** = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (**Rc** field, bit 31, in the instruction encoding = '1') to be an invalid form.

Table 2-19 summarizes the integer store instructions.

Table 2-19. Integer Store Instructions

Name	Mnemonic	Syntax
Store Byte	stb	rS,d(rA)
Store Byte Indexed	stbx	rS,rA,rB
Store Byte with Update	stbu	rS,d(rA)
Store Byte with Update Indexed	stbux	rS,rA,rB
Store Halfword	sth	rS,d(rA)
Store Halfword Indexed	sthx	rS,rA,rB
Store Halfword with Update	sthu	rS,d(rA)
Store Halfword with Update Indexed	sthux	rS,rA,rB
Store Word	stw	rS,d(rA)
Store Word Indexed	stwx	rS,rA,rB
Store Word with Update	stwu	rS,d(rA)
Store Word with Update Indexed	stwux	rS,rA,rB

Integer Load and Store with Byte-Reverse Instructions

Table 2-20 describes integer load and store with byte-reverse instructions. When used in a PowerPC system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a PowerPC system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see the byte ordering section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 2-20. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax
Load Halfword Byte-Reverse Indexed	lhbrx	rD,rA,rB
Load Word Byte-Reverse Indexed	lwbrx	rD,rA,rB
Store Halfword Byte-Reverse Indexed	sthbrx	rS,rA,rB
Store Word Byte-Reverse Indexed	stwbrx	rS,rA,rB

Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions can have operands that require memory accesses crossing a 4 KB page boundary. As a result, these instructions might be interrupted by a DSI exception associated with the address translation of the second page.

Implementation Notes: The following list describes the Espresso implementation of the load and store multiple instruction:

- For load/store string operations, the hardware does not combine register values to reduce the number of discrete accesses. However, if store gathering is enabled and the accesses fall under the criteria for store gathering, the stores can be combined to enhance performance. At a minimum, additional cache access cycles are required.
- The Espresso implementation supports misaligned, single-register load and store accesses in little-endian mode without causing an alignment exception. However, execution of misaligned load/store multiple/string operations causes an alignment exception.

The PowerPC Architecture defines the load multiple word (**lmw**) instruction with **rA** in the range of registers to be loaded as an invalid form. *Table 2-21* describes the load/store multiple instructions.

Table 2-21. Integer Load/Store Multiple Instructions

Name	Mnemonic	Syntax
Load Multiple Word	lmw	rD,d(rA)
Store Multiple Word	stmw	rS,d(rA)

Integer Load and Store String Instructions

The integer load/store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Table 2-22 summarizes the integer load/store string instructions. In other PowerPC implementations operating with little-endian byte order, execution of a load or string instruction invokes the alignment error handler; see the byte ordering section of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

Table 2-22. Integer Load/Store String Instructions

Name	Mnemonic	Syntax
Load String Word Immediate	lswi	rD,rA,NB
Load String Word Indexed	lswx	rD,rA,rB
Store String Word Immediate	stswi	rS,rA,NB
Store String Word Indexed	stswx	rS,rA,rB

Load string and store string instructions can involve operands that are not word-aligned.

As described in *Section 4.5.5 Alignment Exception (x'00600')* on page 147, a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type.

A non-word-aligned string operation that crosses a 4 KB boundary or a word-aligned string operation that crosses a 256 MB boundary always causes an alignment exception. A non-word-aligned string operation that crosses a doubleword boundary is also slower than a word-aligned string operation.

Implementation Note: Espresso implements the load/store string instructions as follows:

- For load and store string operations, the hardware does not combine register values to reduce the number of discrete accesses. However, if store gathering is enabled and the accesses fall under the criteria for store gathering, the stores can be combined to enhance performance. At a minimum, additional cache access cycles are required.
- The Espresso implementation supports misaligned, single-register load and store accesses in little-endian mode without causing an alignment exception. However, execution of misaligned load/store multiple/string operations causes an alignment exception.

Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode. Floating-point loads and stores are not supported for direct-store accesses. The use of floating-point loads and stores for direct-store access results in an alignment exception.

Implementation Notes: The Espresso implementation treats exceptions as follows:

- The FPU can be run in two different modes: ignore exceptions mode (MSR[FE0] = MSR[FE1] = '0') and precise mode (any other settings for MSR[FE0,FE1]). For the Espresso implementation, ignore excep-

tions mode allows floating-point instructions to complete earlier and thus can provide better performance than precise mode.

- The floating-point load and store indexed instructions (**lfsx**, **lfsux**, **lfdx**, **lfdux**, **stfsx**, **stfsux**, **stfdx**, **stfdux**) are invalid when the Rc bit is one. In the Espresso implementation, executing one of these invalid instruction forms causes CR0 to be set to an undefined value.

Floating-Point Load Instructions

There are three forms of the floating-point load instruction: single-precision, double-precision, and paired single (quantized) operand formats. The behavior of double-precision floating-point load instructions and the behavior of single-precision floating-point load instructions when HID2[PSE] = '0' are described here. Paired single floating-point load instructions are illegal when HID2[PSE] = '0'. The behavior of single-precision floating-point load instructions and paired single (quantized) load instructions when HID2[PSE] = '1' are described in *Paired Single Load and Store Instructions* on page 92.

Single-precision floating-point load instructions convert single-precision data to double-precision format before loading an operand into an FPR.

The PowerPC Architecture defines a load with update instruction with rA = 0 as an invalid form.

Table 2-23 summarizes the single-precision and double-precision floating-point load instructions.

Table 2-23. Floating-Point Load Instructions

Name	Mnemonic	Syntax
Load Floating-Point Single	lfs	frD,d(rA)
Load Floating-Point Single Indexed	lfsx	frD,rA,rB
Load Floating-Point Single with Update	lfsu	frD,d(rA)
Load Floating-Point Single with Update Indexed	lfsux	frD,rA,rB
Load Floating-Point Double	lfd	frD,d(rA)
Load Floating-Point Double Indexed	lfdx	frD,rA,rB
Load Floating-Point Double with Update	lfd u	frD,d(rA)
Load Floating-Point Double with Update Indexed	lfd ux	frD,rA,rB

Floating-Point Store Instructions

There are four basic forms of the store instruction: single-precision, double-precision, paired single (quantized), and integer. The integer form is supported by the optional **stfiwx** instruction. The behavior of double-precision floating-point store instructions and the behavior of single-precision floating-point store instructions when HID2[PSE] = '0' are described here. Paired single floating-point store instructions are illegal when HID2[PSE] = '0'. The behavior of single-precision floating-point store instructions and paired single (quantized) store instructions when HID2[PSE] = '1' is described in *Paired Single Load and Store Instructions* on page 92. Single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands.

Implementation Note: The 32 floating-point registers must be initialized after POR by loading any value from memory, using an **lfd** instruction. This puts the internal representation of each FPR into a valid state. It is necessary to ensure that an FPR is in a valid state before attempting to store the value to memory using a **stfd** instruction.

Table 2-24 summarizes the single-precision and double-precision floating-point store and **stfiwx** instructions. Some floating-point store instructions require conversions in the LSU.

Table 2-24. Floating-Point Store Instructions

Name	Mnemonic	Syntax
Store Floating-Point Single	stfs	frS,d(rA)
Store Floating-Point Single Indexed	stfsx	frS,rB
Store Floating-Point Single with Update	stfsu	frS,d(rA)
Store Floating-Point Single with Update Indexed	stfsux	frS,rB
Store Floating-Point Double	stfd	frS,d(rA)
Store Floating-Point Double Indexed	stfdx	frS,rB
Store Floating-Point Double with Update	stfdu	frS,d(rA)
Store Floating-Point Double with Update Indexed	stfdux	frS,rB
Store Floating-Point as Integer Word Indexed ¹	stfiwx	frS,rB

1. The **stfiwx** instruction is optional in the PowerPC Architecture.

Table 2-25 shows conversions that the LSU makes when executing a Store Floating-Point Single instruction (when **HID2[PSE]** = '0').

Table 2-25. Store Floating-Point Single Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	If ($\text{exp} \leq 896$) then Denormalize and Store else Store
Double	Denormalized	Store zero
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Note: The FPRs are not initialized by **HRESET**, and they must be initialized with some valid value after POR **HRESET** and before being stored.

Table 2-26 shows the conversions made when performing a Store Floating-Point Double instruction. Most entries in the table indicate that the floating-point value is simply stored. Only in a few cases are any other actions taken.

Table 2-26. Store Floating-Point Double Behavior

FPR Precision	Data Type	Action
Single	Normalized	Store
Single	Denormalized	Normalize and Store
Single	Zero, infinity, QNaN	Store
Single	SNaN	Store
Double	Normalized	Store
Double	Denormalized	Store
Double	Zero, infinity, QNaN	Store
Double	SNaN	Store

Architecturally, all single-precision and double-precision floating-point numbers are represented in double-precision format within Espresso. Execution of a store floating-point single (**stfs**, **stfsu**, **stfsx**, **stfsux**) instruction requires conversion from double-precision to single-precision format. If the exponent is not greater than 896, this conversion requires denormalization. Espresso supports this denormalization by shifting the mantissa one bit at a time. Anywhere from 1 - 23 clock cycles are required to complete the denormalization, depending upon the value to be stored.

Because of how floating-point numbers are implemented in Espresso, there is also a case when execution of a store floating-point double (**stfd**, **stfdu**, **stfdx**, **stfdux**) instruction can require internal shifting of the mantissa. This case occurs when the operand of a store floating-point double instruction is a denormalized single-precision value. The value can be the result of a load floating-point single instruction, a single-precision arithmetic instruction, or a floating round to single-precision instruction. In these cases, shifting the mantissa takes from 1 - 23 clock cycles, depending upon the value to be stored. These cycles are incurred during the store.

Paired Single Load and Store Instructions

In addition to the floating-point load and store instructions defined in the PowerPC Architecture, Espresso includes eight additional load and store instructions that can implicitly convert their operands between single-precision floating-point and lower precision, quantized data types. For load instructions, this conversion is an inverse quantization, or dequantization, operation that converts signed or unsigned, 8- or 16-bit integers to 32-bit single-precision floating-point operands. This conversion takes place in the LSU as the data is being transferred to a floating-point register (FPR). For store instructions, the conversion is a quantization operation that converts single-precision floating-point numbers to operands having one of the quantized data types. This conversion takes place in the LSU as the data is transferred out of an FPR.

The load and store instructions for which data quantization applies are for “paired single” operands, and are valid only when HID2[PSE] = ‘1’. These load and store instructions cause an illegal instruction exception if execution is attempted when HID2[PSE] = ‘0’. Furthermore, the nonindexed forms of these loads and stores (**psq_l[u]** and **psq_st[u]**) are illegal unless HID2[LSQE] = ‘1’ as well. The quantization/dequantization hardware in the LSU assumes big-endian ordering of the data in memory. Use of these instructions in little-endian mode (MSR[LE] = ‘1’) gives undefined results. Whenever a pair of operands are converted, they are both converted in the same manner.

When operating in paired single mode (HID2[PSE] = '1'), the behavior of single-precision floating-point load and store instructions is different from that described in the previous two sections. In this mode, a single-precision, floating-point load instruction loads one single-precision operand into both the high-order and low-order words of the operand pair in an FPR. A single-precision, floating-point store instruction stores only the high-order word of the operand pair in an FPR.

Table 2-27 summarizes the paired single load and store instructions.

Table 2-27. Paired Single Load and Store Instructions

Name	Mnemonic	Syntax
Paired Single Quantized Load ²	psq_l	frD,d(rA),W,qrl
Paired Single Quantized Load Indexed ¹	psq_lx	frD,rA,rB,W,qrl
Paired Single Quantized Load with Update ²	psq_lu	frD,d(rA),W,qrl
Paired Single Quantized Load with Update Indexed ¹	psq_lux	frD,rA,rB,W,qrl
Paired Single Quantized Store ²	psq_st	frS,d(rA),W,qrl
Paired Single Quantized Store Indexed ¹	psq_stx	frS,rA,rB,W,qrl
Paired Single Quantized Store with Update ²	psq_stu	frS,d(rA),W,qrl
Paired Single Quantized Store with Update Indexed ¹	psq_stux	frS,rA,rB,W,qrl

1. These instructions belong to the Espresso graphics extensions and are legal only when HID2[PSE] = '1'.
2. These instructions belong to the Espresso graphics extensions and are legal only when HID2[PSE] = '1' and HID2[LSQE] = '1'.

Two paired single load (**psq_l**, **psq_lu**) and two paired single store (**psq_st**, **psq_stu**) instructions use a variation of the D-form instruction format. Instead of having a 16-bit displacement field, 12 bits are used for displacement. The remaining 4 bits are used to specify whether one or two operands are to be processed (the 1-bit W field) and which of the eight GQRs is to be used to specify the scale and type for the conversion (the 3-bit I field). The two remaining paired single load (**psq_lx**, **psq_lux**) and the two remaining paired single store (**psq_stx**, **psq_stux**) instructions use a variation of the X-form instruction format. Instead of having a 10-bit secondary opcode field, 6 bits are used for the secondary opcode, and the remaining 4 bits are used for the W field and the I field.

See *Section 10 Espresso PowerPC Instruction Set* on page 255 for more information about the instruction format.

The dequantization algorithm used to convert each integer of a pair to a single-precision floating-point operand is as follows:

1. Read the integer operand from the L1 cache.
2. Convert data to sign and magnitude according to the type specified in the selected GQR.
3. Convert the magnitude to the normalized mantissa and exponent.
4. Subtract the scaling factor specified in the selected GQR from the exponent.
5. Load the converted value into the target FPR.

For an integer value, I, in memory, the floating-point value, F, loaded into the target FPR, is $F = I \times 2^{-S}$, where S is the two's complement value in the LD_SCALE field of the selected GQR. Table 2-28 shows how an integer value of 1 is converted to a single-precision floating-point value for various scaling factors.

Table 2-28. Conversion of Integer Value 1 to Single-Precision Floating-Point Value

GQRx[LD_SCALE]	Scaling Factor (S)	Floating-Point Value
100000	-32	4.29 E+9
100001	-31	2.15 E+9
...		
111110	-2	4.00 E+0
111111	-1	2.00 E+0
000000	0	1.00 E+0
000001	1	5.00 E-1
000010	2	2.50 E-1
...		
011110	30	9.31 E-10
011111	31	4.66 E-10

For a single-precision floating-point operand (type = 0), the value from the L1 cache is passed directly to the register without any conversion. This includes the case where the operand is a denorm.

The quantization algorithm used to convert each single-precision floating-point operand of a pair to an integer is as follows:

1. Move the single-precision floating-point operand from the FPR to the completion store queue.
2. Add the scaling factor specified in the selected GQR to the exponent.
3. Shift the mantissa and increment or decrement the exponent until it is zero.
4. Convert the sign and magnitude to the twos complement representation.
5. Round toward zero to get the type specified in the selected GQR.
6. Adjust the resulting value on overflow.
7. Store the converted value in the L1 cache.

The adjusted result value for overflow of unsigned integers is zero for negative values. For positive values, the adjusted result value for overflow is 255 for 8-bit types and 65535 for 16-bit types. The adjusted result value for overflow of signed negative integers is -128 for 8-bit types and -32768 for 16-bit types. The adjusted result value for overflow of signed positive integers is 127 for 8-bit types and 32767 for 16-bit types. The converted value produced when the input operand is +Inf or a NaN with a '0' in the high-order bit is the same as the adjusted result value for overflow of positive values for the target data type. The converted value produced when the input operand is -Inf or a NaN with a '1' in the high-order bit is the same as the adjusted result value for overflow of negative values.

For a single-precision floating-point value, F , in an FPR, the integer value I , stored to memory, is $I = \text{ROUND}(F \times 2^S)$, where S is the twos complement value in the ST_SCALE field of the selected GQR, and ROUND applies the rounding and clamping appropriate to the particular target integer format.

Table 2-29 shows how a floating-point value of 1.00 E+2 is converted to an integer value for various scaling factors.

Table 2-29. Conversion of Floating-Point Value 1.00 E+2 to Integer

GQRx[LD_SCALE]	Scaling Factor (S)	u8 Value	u16	s8	s16
100000	-32	0	0	0	0
100001	-31	0	0	0	0
...					
111110	-2	25	25	25	25
111111	-1	50	50	50	50
000000	0	100	100	100	100
000001	1	200	200	127	200
000010	2	255	400	127	400
...					
011110	30	255	65535	127	32767
011111	31	255	65525	127	32767

For a single-precision floating-point operand (type = 0), the value from the FPR is passed directly to the L1 cache without any conversion, except when this operand is a denorm. In the case of a denorm, the value 0.0 is stored in the L1 cache.

2.3.4.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress can affect the particular CR bit. If no interlock is found, the branch can be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the PowerPC processors ignore the 2 low-order bits of the generated branch target address.

Branch instructions compute the EA of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

Note: In Espresso, all branch instructions (**b**, **ba**, **bl**, **bla**, **bc**, **bca**, **bcl**, **bcla**, **bclr**, **bclrl**, **bcctr**, **bcctrl**) and condition register logical instructions (**crand**, **cror**, **crxor**, **crnand**, **crnor**, **crandc**, **creqv**, **crorc**, and **mcrf**) are executed by the BPU. Some of these instructions can redirect instruction execution conditionally based on the value of bits in the CR. Whenever the CR bits resolve, the branch direction is either marked as correct or mispredicted. Correcting a mispredicted branch requires Espresso to flush speculatively executed instructions and restore the machine state to immediately after the branch. This correction can be done immediately upon resolution of the Condition Register bits.

Branch Instructions

Table 2-30 lists the branch instructions provided by the PowerPC processors. To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions.

See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a list of simplified mnemonic examples.

Table 2-30. Branch Instructions

Name	Mnemonic	Syntax
Branch	b (ba bl bla)	target_addr
Branch Conditional	bc (bca bcl bcla)	BO,BI,target_addr
Branch Conditional to Link Register	bclr (bclrl)	BO,BI
Branch Conditional to Count Register	bcctr (bcctrl)	BO,BI

Condition Register Logical Instructions

Condition register logical instructions and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Table 2-31 lists these instructions.

Table 2-31. Condition Register Logical Instructions

Name	Mnemonic	Syntax
Condition Register AND	crand	crbD,crbA,crbB
Condition Register OR	cror	crbD,crbA,crbB
Condition Register XOR	crxor	crbD,crbA,crbB
Condition Register NAND	crnand	crbD,crbA,crbB
Condition Register NOR	crnor	crbD,crbA,crbB
Condition Register Equivalent	creqv	crbD,crbA,crbB
Condition Register AND with Complement	crandc	crbD,crbA,crbB
Condition Register OR with Complement	crorc	crbD,crbA,crbB
Move Condition Register Field	mcrf	crfD,crfS

Note: If the LR update option is enabled for any of these instructions, the PowerPC Architecture defines these forms of the instructions as invalid.

Trap Instructions

The trap instructions shown in Table 2-32 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap type program exception is taken. For more information, see Section 4.5.6 Program Exception (x'00700') on page 147. If the tested conditions are not met, instruction execution continues normally.

Table 2-32. Trap Instructions

Name	Mnemonic	Syntax
Trap Word Immediate	twi	TO,rA,SIMM
Trap Word	tw	TO,rA,rB

See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete set of simplified mnemonics.

2.3.4.5 System Linkage Instruction (UISA)

The System Call (**sc**) instruction, shown in *Table 2-33*, permits a program to call on the system to perform a service. See *Section 2.3.7.1* on page 108 for additional information.

Table 2-33. System Linkage Instruction (UISA)

Name	Mnemonic	Syntax
System Call	sc	—

Executing this instruction causes the system call exception handler to be evoked. For more information, see *Section 4.5.9 System Call Exception (x'00C00')* on page 148.

2.3.4.6 Processor Control Instructions (UISA)

Processor control instructions are used to read from and write to the Condition Register (CR), machine state register (MSR), and special-purpose registers (SPRs).

See *Section 2.3.6 Processor Control Instructions (VEA)* on page 104 for the **mftb** instruction and *Section 2.3.7.2 Processor Control Instructions (OEA)* on page 109 for information about the instructions used for reading from and writing to the MSR and SPRs.

Move to/from Condition Register Instructions

Table 2-34 summarizes the instructions for reading from or writing to the Condition Register.

Table 2-34. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax
Move to Condition Register Fields	mtcrf	CRM,rS
Move to Condition Register from XER	mcrxr	crfD
Move from Condition Register	mfcrr	rD

Implementation Note: The PowerPC Architecture indicates that in some implementations the Move to Condition Register Fields (**mtcrf**) instruction can perform more slowly when only a portion of the fields are updated as opposed to all of the fields. The Condition Register access latency for Espresso is the same in both cases.

Move to/from Special-Purpose Register Instructions (UISA)

Table 2-35 lists the **mtspr** and **mfspir** instructions.

Table 2-35. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	mtspr	SPR,rS
Move from Special-Purpose Register	mfspir	rD,SPR

Table 2-36 lists the SPR numbers for both user-level and supervisor-level accesses.

Table 2-36. SPR Encodings for PowerPC-Defined Registers (Page 1 of 2)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5:9]	spr[0:4]		
CTR	9	00000	01001	User (UISA)	Both
DABR	1013	11111	10101	Supervisor (OEA)	Both
DAR	19	00000	10011	Supervisor (OEA)	Both
DBAT0L	537	10000	11001	Supervisor (OEA)	Both
DBAT0U	536	10000	11000	Supervisor (OEA)	Both
DBAT1L	539	10000	11011	Supervisor (OEA)	Both
DBAT1U	538	10000	11010	Supervisor (OEA)	Both
DBAT2L	541	10000	11101	Supervisor (OEA)	Both
DBAT2U	540	10000	11100	Supervisor (OEA)	Both
DBAT3L	543	10000	11111	Supervisor (OEA)	Both
DBAT3U	542	10000	11110	Supervisor (OEA)	Both
DBAT4L	569	10001	11001	Supervisor (OEA)	Both
DBAT4U	568	10001	11000	Supervisor (OEA)	Both
DBAT5L	571	10001	11011	Supervisor (OEA)	Both
DBAT5U	570	10001	11010	Supervisor (OEA)	Both
DBAT6L	573	10001	11101	Supervisor (OEA)	Both
DBAT6U	572	10001	11100	Supervisor (OEA)	Both
DBAT7L	575	10001	11111	Supervisor (OEA)	Both
DBAT7U	574	10001	11110	Supervisor (OEA)	Both
DEC	22	00000	10110	Supervisor (OEA)	Both
DSISR	18	00000	10010	Supervisor (OEA)	Both
EAR	282	01000	11010	Supervisor (OEA)	Both
IBAT0L	529	10000	10001	Supervisor (OEA)	Both
IBAT0U	528	10000	10000	Supervisor (OEA)	Both
IBAT1L	531	10000	10011	Supervisor (OEA)	Both
IBAT1U	530	10000	10010	Supervisor (OEA)	Both
IBAT2L	533	10000	10101	Supervisor (OEA)	Both
IBAT2U	532	10000	10100	Supervisor (OEA)	Both

1. The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.
2. The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** or **mfspr** instruction and specifying TBR 268 for TBL and SPR 269 for TBU.
3. This is an Espresso register that has been moved out of the core and is treated like a global register.

Table 2-36. SPR Encodings for PowerPC-Defined Registers (Page 2 of 2)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5:9]	spr[0:4]		
IBAT3L	535	10000	10111	Supervisor (OEA)	Both
IBAT3U	534	10000	10110	Supervisor (OEA)	Both
IBAT4L	561	10001	10001	Supervisor (OEA)	Both
IBAT4U	560	10001	10000	Supervisor (OEA)	Both
IBAT5L	563	10001	10011	Supervisor (OEA)	Both
IBAT5U	562	10001	10010	Supervisor (OEA)	Both
IBAT6L	565	10001	10101	Supervisor (OEA)	Both
IBAT6U	564	10001	10100	Supervisor (OEA)	Both
IBAT7L	567	10001	10111	Supervisor (OEA)	Both
IBAT7U	566	10001	10110	Supervisor (OEA)	Both
LR	8	00000	01000	User (UISA)	Both
PVR ³	287	01000	11111	Supervisor (OEA)	mfspr
SDR1	25	00000	11001	Supervisor (OEA)	Both
SPRG0	272	01000	10000	Supervisor (OEA)	Both
SPRG1	273	01000	10001	Supervisor (OEA)	Both
SPRG2	274	01000	10010	Supervisor (OEA)	Both
SPRG3	275	01000	10011	Supervisor (OEA)	Both
SRR0	26	00000	11010	Supervisor (OEA)	Both
SRR1	27	00000	11011	Supervisor (OEA)	Both
TBL ^{2,3}	268	01000	01100	User (VEA)	mfspr
	284	01000	11100	Supervisor (OEA)	mtspr
TBU ^{2,3}	269	01000	01101	User (VEA)	mfspr
	285	01000	11101	Supervisor (OEA)	mtspr
XER	1	00000	00001	User (UISA)	Both

1. The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.
2. The TB registers are referred to as TBRs rather than SPRs and can be written to using the **mtspr** instruction in supervisor mode and the TBR numbers here. The TB registers can be read in user mode using either the **mftb** or **mfspr** instruction and specifying TBR 268 for TBL and SPR 269 for TBU.
3. This is an Espresso register that has been moved out of the core and is treated like a global register.

Encodings for the Espresso-specific SPRs are listed in *Table 2-37* on page 101.

Table 2-37. SPR Encodings for Espresso-Defined Registers (Page 1 of 2)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5:9]	spr[0:4]		
DMAL ²	923	11100	11011	Supervisor	Both
DMAU ²	922	11100	11010	Supervisor	Both
GQR0 ²	912	11100	10000	Supervisor	Both
GQR1 ²	913	11100	10001	Supervisor	Both
GQR2 ²	914	11100	10010	Supervisor	Both
GQR3 ²	915	11100	10011	Supervisor	Both
GQR4 ²	916	11100	10100	Supervisor	Both
GQR5 ²	917	11100	10101	Supervisor	Both
GQR6 ²	918	11100	10110	Supervisor	Both
GQR7 ²	919	11100	10111	Supervisor	Both
IABR	1010	11111	10010	Supervisor	Both
ICTC	1019	11111	11011	Supervisor	Both
L2CR	1017	11111	11001	Supervisor	Both
MMCR0	952	11101	11000	Supervisor	Both
MMCR1	956	11101	11100	Supervisor	Both
PCSR ³	946	11101	10010	Supervisor	Both
UPIR	1007	11111	01111	User	mfspr
PIR	1023	11111	11111	Supervisor	mfspr
PMC1	953	11101	11001	Supervisor	Both
PMC2	954	11101	11010	Supervisor	Both
PMC3	957	11101	11101	Supervisor	Both
PMC4	958	11101	11110	Supervisor	Both
SIA	955	11101	11011	Supervisor	Both
UDMAL ²	907	11100	01011	User	Both
UDMAU ²	906	11100	01010	User	Both
UGQR0 ²	896	11100	00000	User	Both
UGQR1 ²	897	11100	00001	User	Both
UGQR2 ²	898	11100	00010	User	Both

Note:

1. The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.
For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.
2. This register is part of the Espresso graphics extensions.
3. This register is global to the Espresso chip. The single register is shared among the three cores. The register should be written by only one core at any given time, but can be read by any core at any time.

Table 2-37. SPR Encodings for Espresso-Defined Registers (Page 2 of 2)

Register Name	SPR ¹			Access	mfspr/mtspr
	Decimal	spr[5:9]	spr[0:4]		
UGQR3 ²	899	11100	00011	User	Both
UGQR4 ²	900	11100	00100	User	Both
UGQR5 ²	901	11100	00101	User	Both
UGQR6 ²	902	11100	00110	User	Both
UGQR7 ²	903	11100	00111	User	Both
UHD2 ²	904	11100	01000	User	mfspr
UMMCR0	936	11101	01000	User	mfspr
UMMCR1	940	11101	01100	User	mfspr
UPMC1	937	11101	01001	User	mfspr
UPMC2	938	11101	01010	User	mfspr
UPMC3	941	11101	01101	User	mfspr
UPMC4	942	11101	01110	User	mfspr
USIA	939	11101	01011	User	mfspr
UWPAR ²	905	11100	01001	User	mfspr

Note:

1. The order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.
For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order 5 bits appearing in bits 16:20 of the instruction and the low-order 5 bits in bits 11:15.
2. This register is part of the Espresso graphics extensions.
3. This register is global to the Espresso chip. The single register is shared among the three cores. The register should be written by only one core at any given time, but can be read by any core at any time.

2.3.4.7 Memory Synchronization Instructions (UISA)

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events and the order in which memory operations are seen by other processors or memory access mechanisms. See *Section 3 Espresso Instruction and Data Cache Operation* on page 111 for additional information about these instructions and about related aspects of memory synchronization. *Table 2-38* on page 103 lists a summary of the memory synchronization instructions.

Architectural note: The architecture emphasizes that a store instruction to the cache line addressed by a **lwarx/stwcx**. pair can cause a livelock if included in the **lwarx/stwcx**. loop (see Appendix E of the *PowerPC Microprocessor Family: The Programming Environments* manual for more information). In Espresso, such a store within the **lwarx/stwcx**. loop must be avoided.

Table 2-38. Memory Synchronization Instructions (UISA)

Name	Mnemonic	Syntax	Implementation Notes
Load Word and Reserve Indexed	lwarx	rD,rA,rB	Programmers can use lwarx with stwcx . to emulate common semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add. Both instructions must use the same EA. Reservation granularity is implementation-dependent. Espresso makes reservations on behalf of aligned 32-byte sections of the memory address space. If the W bit is set, executing lwarx and stwcx . to a page marked write-through does not cause a DSI exception, but DSI exceptions can result for other reasons. If the location is not word-aligned, an alignment exception occurs.
Store Word Conditional Indexed	stwcx.	rS,rA,rB	The stwcx. instruction is the only load/store instruction with a valid form if Rc is set. If Rc is zero, executing stwcx. sets CR0 to an undefined value. In general, stwcx. always causes a transaction on the external bus and thus operates with slightly worse performance characteristics than normal store operations. For proper multiprocessor execution, the cache line addressed by these instructions must be in a cache-inhibited address space. Alternatively, a dcbf instruction must precede every instance of the stwcx. instruction.
Synchronize	sync	—	Because it delays subsequent instructions until all previous instructions complete to where they cannot cause an exception, sync is a barrier against store gathering when HID2[LCE] = '0' and HID2[WPE] = '0'. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of the modified sync behavior when HID2[LCE] = '1' or HID2[WPE] = '1'. Additionally, all load/store cache/bus activities initiated by prior instructions are completed. Touch load operations (dcbt , dcbtst) must complete address translation, but need not complete on the bus. The sync operation completes after a successful broadcast. The latency of sync depends on the processor state when it is dispatched and on various system-level situations. Therefore, frequent use of sync can degrade performance.

See *Section 2.3.6.1 Memory Synchronization Instructions (VEA)* on page 104 for details about additional memory synchronization (**ieio** and **isync**) instructions.

In the PowerPC Architecture, the Rc bit must be zero for most load and store instructions. If Rc is set, the instruction form is invalid for **sync** and **lwarx** instructions. If Espresso encounters one of these invalid instruction forms, it sets CR0 to an undefined value.

2.3.5 PowerPC VEA Instructions

The PowerPC virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache control instructions, address aliasing, and other related issues. Implementations that conform to the VEA also adhere to the UISA, but might not necessarily adhere to the OEA.

This section describes additional instructions that are provided by the VEA.

2.3.6 Processor Control Instructions (VEA)

In addition to the Move to Condition Register instructions (specified by the UISA), the VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the Time Base Register; see *Section 3 Espresso Instruction and Data Cache Operation* on page 111 for more information.

Table 2-39 shows the **mftb** instruction.

Table 2-39. Move from Time Base Instruction

Name	Mnemonic	Syntax
Move from Time Base	mftb	rD, TBR

Simplified mnemonics are provided for the **mftb** instruction so that it can be coded with the TBR name as part of the mnemonic rather than requiring it to be coded as an operand. See the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual for simplified mnemonic examples and for simplified mnemonics for Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**), which are variants of the **mftb** instruction rather than of **mfsprr**. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form and an **mftb** mnemonic with one operand as the simplified form. Note that Espresso ignores the extended opcode differences between **mftb** and **mfsprr** by ignoring bit 25 and treating both instructions identically.

Implementation Notes:

- Espresso allows user-mode read access to the time base counter through the use of the Move from Time Base (**mftb**) and the Move from Time Base Upper (**mftbu**) instructions. As a 32-bit PowerPC implementation, Espresso can access TBU and TBL only separately, whereas 64-bit implementations can access the entire TB register at once.
- The time base counter is clocked at a frequency that is one-fourth that of the bus clock.

2.3.6.1 Memory Synchronization Instructions (VEA)

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. See *Section 3 Espresso Instruction and Data Cache Operation* on page 111 for more information about these instructions and about related aspects of memory synchronization.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eieio**) and Instruction Synchronize (**isync**) instructions. The number of cycles required to complete an **eieio** instruction depends on system parameters and on the processor state when the instruction is issued. As a result, frequent use of this instruction can degrade performance slightly.

Table 2-40 describes the memory synchronization instructions defined by the VEA.

Table 2-40. Memory Synchronization Instructions (VEA)

Name	Mnemonic	Syntax	Implementation Notes
Enforce In-Order Execution of I/O	eieio	—	The eieio instruction is dispatched to the LSU and executes after all previous cache-inhibited or write-through accesses are performed; all subsequent instructions that generate such accesses execute after eieio. EIEIO operation is broadcast on the external bus to enforce ordering in the external memory system. The eieio operation bypasses the L2 cache and is forwarded to the bus unit. Because Espresso does not reorder noncacheable accesses, eieio is not needed to force ordering. However, if store gathering is enabled and an eieio is detected in a store queue, stores are not gathered. Broadcasting eieio prevents external devices, such as a bus bridge chip, from gathering stores. The behavior of eieio is modified when either $HID2[LCE] = '1'$ or $HID2[WPE] = '1'$. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of this modified behavior.
Instruction Synchronize	isync	—	The isync instruction is refetch serializing; that is, it causes Espresso to purge its instruction queue and wait for all prior instructions to complete before refetching the next instruction, which is not executed until all previous instructions complete to the point where they cannot cause an exception. The isync instruction does not wait for all pending stores in the store queue to complete. Any instruction after an isync sees all effects of prior instructions.

2.3.6.2 Memory Control Instructions (VEA)

Memory control instructions can be classified as follows:

- Cache management instructions (user-level and supervisor-level)
- Segment register manipulation instructions (OEA)
- Translation lookaside buffer management instructions (OEA)

This section describes the user-level cache management instructions defined by the VEA. See *Section 2.3.7.3* on page 109 for information about supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

User-Level Cache Instructions (VEA)

The instructions summarized in this section help user-level programs manage on-chip caches if they are implemented. See *Section 3 Espresso Instruction and Data Cache Operation* on page 111 for more information about cache topics. The following sections describe how these operations are treated with respect to the Espresso cache.

As with other memory-related instructions, the effects of cache management instructions on memory are weakly-ordered. If the programmer must ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed after those instructions.

The Espresso cores broadcast and snoop all cache control operations among themselves. All cache control instructions to T = '1' space are no-ops. For information about how cache control instructions affect the L2 cache, see *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229.

Table 2-41 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 2-41. User-Level Cache Instructions (Page 1 of 2)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	dcbi	rA,rB	The dcbi instruction is typically a supervisor-level instruction but can be made a user-level instruction.
Data Cache Block Touch ¹	dcbt	rA,rB	The VEA defines this instruction to allow for potential system performance enhancements through the use of software-initiated prefetch hints. Implementations are not required to take any action based on execution of this instruction, but they can prefetch the cache block corresponding to the EA into their cache. When dcbt executes, Espresso checks for protection violations (as for a load instruction). This instruction is treated as a no-op for the following cases: • A valid translation is not found either in the BAT or TLB. • The access causes a protection violation. • The page is mapped cache-inhibited, G = '1' (guarded), or T = '1'. • The cache is locked or disabled. • HID0[NOOPTI] = '1'. Otherwise, if no data is in the cache location, Espresso requests a cache line fill with intent to modify. Data brought into the cache is validated as if it were a load instruction. The memory reference of a dcbt sets the reference bit. The behavior of dcbt is modified when either HID2[LCE] = '1' or HID2[WPE] = '1'. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of this modified behavior.
Data Cache Block Touch for Store ¹	dcbtst	rA,rB	This instruction behaves like dcbt .
Data Cache Block Set to Zero	dcbz	rA,rB	The EA is computed, translated, and checked for protection violations. For cache hits, 4 beats of zeros are written to the cache block and the tag is marked M. For cache misses with the replacement block marked E, the zero line fill is performed and the cache block is marked M. However, if the replacement block is marked M, the contents are written back to memory first. The instruction executes regardless of whether the cache is locked; if the cache is disabled, an alignment exception occurs. If M = '1' (coherency enforced), the address is broadcast to the bus before the zero line fill. The exception priorities (from highest to lowest) are as follows: 1 Cache disabled—Alignment exception 2 Page marked write-through or cache Inhibited—Alignment exception 3 BAT protection violation—DSI exception 4 TLB protection violation—DSI exception A dcbz is broadcast on the bus. The behavior of dcbz is modified when either HID2[LCE] = '1' or HID2[WPE] = '1'. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of this modified behavior.
Data Cache Block Set to Zero Locked	dcbz_l	rA,rB	This instruction is illegal when HID2[LCE] = '0'. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of this instruction when HID2[LCE] = '1'. Note: The dcbz_l is user-level by default and can be set to supervisor-level.
1. A program that uses dcbt and dcbtst instructions improperly performs less efficiently. To improve performance, HID0[NOOPTI] can be set, which causes dcbt and dcbtst to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero, which enables the use of these instructions.			

Table 2-41. User-Level Cache Instructions (Page 2 of 2)

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Store	dcbst	rA,rB	The EA is computed, translated, and checked for protection violations. - For cache hits with the tag marked E, no further action is taken. - For cache hits with the tag marked M, the cache block is written back to memory and marked E. A dcbst is broadcast on the bus. The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for dcbst are as follows: - BAT protection violation - DSI exception - TLB protection violation - DSI exception The behavior of dcbst is modified when either HID2[LCE] = '1' or HID2[WPE] = '1'. See <i>Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe</i> on page 229 for a description of this modified behavior.
Data Cache Block Flush	dcbf	rA,rB	The EA is computed, translated, and checked for protection violations. - For cache hits with the tag marked M, the cache block is written back to memory and the cache entry is invalidated. - For cache hits with the tag marked E, the entry is invalidated. - For cache misses, no further action is taken. A dcbf is broadcast on the bus. The instruction acts like a load with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for dcbf are as follows: - BAT protection violation - DSI exception - TLB protection violation - DSI exception The behavior of dcbf is modified when either HID2[LCE] = '1' or HID2[WPE] = '1'. See <i>Section 8</i> on page 229 for a description of this modified behavior.
Instruction Cache Block Invalidate	icbi	rA,rB	This instruction performs a virtual lookup into the instruction cache (index only). The address is not translated; therefore, it cannot cause an exception. All ways of a selected set are invalidated regardless of whether the cache is disabled or locked. An icbi is broadcast on the bus.
1. A program that uses dcbt and dcbtst instructions improperly performs less efficiently. To improve performance, HID0[NOOPTI] can be set, which causes dcbt and dcbtst to be no-oped at the cache. They do not cause bus activity and cause only a 1-clock execution latency. The default state of this bit is zero, which enables the use of these instructions.			

2.3.6.3 Optional External Control Instructions

The PowerPC Architecture defines an optional external control feature that, if implemented, is supported by the two external control instructions, **eciwx** and **ecowx**. These instructions allow a user-level program to communicate with a special-purpose device. These instructions are summarized in *Table 2-42*.

Table 2-42. External Control Instructions

Name	Mnemonic	Syntax	Implementation Notes
External Control In Word Indexed	eciwx	rD,rA,rB	A transfer size of 4 bytes is implied; the $\overline{\text{TBST}}$ and $\text{TSIZ}[0-2]$ signals are redefined to specify the Resource ID (RID), copied from bits $\text{EAR}[28-31]$. For these operations, $\overline{\text{TBST}}$ carries the $\text{EAR}[28]$ data. Misaligned operands for these instructions cause an alignment exception. Addressing a location where $\text{SR}[\text{T}] = '1'$ causes a DSI exception. If $\text{MSR}[\text{DR}] = 0$, a programming error occurs and the physical address on the bus is undefined. Note: These instructions are optional to the PowerPC Architecture.
External Control Out Word Indexed	ecowx	rS,rA,rB	

The **eciwx/ecowx** instructions enable a system designer to map special devices in an alternative way. The MMU translation of the EA is not used to select the special device, because it is used in most instructions such as loads and stores. Rather, it is used as an address operand that is passed to the device over the address bus. Four other signals (the burst and size signals on the 60Xe bus) are used to select the device; these four signals output the 4-bit resource ID (RID) field located in the EAR. The **eciwx** instruction also loads a word from the data bus that is output by the special device.

2.3.7 PowerPC OEA Instructions

The PowerPC operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the exception model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA.

2.3.7.1 System Linkage Instructions (OEA)

Table 2-43 lists the system linkage instructions. The user-level **sc** instruction lets a user program call on the system to perform a service and causes the processor to take a system call exception. The supervisor-level **rfi** instruction is used for returning from an exception handler.

Table 2-43. System Linkage Instructions (OEA)

Name	Mnemonic	Syntax	Implementation Notes
System Call	sc	—	The sc instruction is context-synchronizing.
Return from Interrupt	rfi	—	The rfi instruction is context-synchronizing. For the Espresso implementation, this means the rfi instruction works its way to the final stage of the execution pipeline, updates architected registers, and redirects the instruction flow.

2.3.7.2 Processor Control Instructions (OEA)

Table 2-44 lists the processor control instructions used to access the MSR.

Table 2-44. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax
Move to Machine State Register	mtmsr	rS
Move from Machine State Register	mfmsr	rD

The OEA defines encodings of **mtspr** and **mfspir** to provide access to supervisor-level registers. Table 2-45 lists the instructions to access the SPRs.

Table 2-45. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Syntax
Move to Special-Purpose Register	mtspr	SPR,rS
Move from Special-Purpose Register	mfspir	rD,SPR

Encodings for the architecture-defined SPRs are listed in Table 2-37 on page 101. Encodings for Espresso-specific, supervisor-level SPRs are listed in Table 2-38 on page 103. Simplified mnemonics are provided for **mtspir** and **mfspir** in the simplified mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

For a description about context synchronization requirements when altering certain SPRs, see the synchronization programming examples section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

2.3.7.3 Memory Control Instructions (OEA)

Memory control instructions include the following:

- Cache management instructions (supervisor-level and user-level)
- Segment register manipulation instructions
- Translation lookaside buffer management instructions

This section describes supervisor-level memory control instructions. Section 2.3.6.2 on page 105 describes user-level memory control instructions.

Supervisor-Level Cache Management Instruction (OEA)

Table 2-46 lists the supervisor-level cache management instructions.

Table 2-46. Supervisor-Level Cache Management Instruction

Name	Mnemonic	Syntax	Implementation Notes
Data Cache Block Invalidate	dcbi	rA,rB	The EA is computed, translated, and checked for protection violations. For cache hits, the cache block is marked I regardless of whether it was marked E or M. A dcbi is broadcast on the bus. The instruction acts like a store with respect to address translation and memory protection. It executes regardless of whether the cache is disabled or locked. The exception priorities (from highest to lowest) for dcbi are as follows: 1 BAT protection violation - DSI exception 2 TLB protection violation - DSI exception Note: The behavior of dcbi is modified when either HID2[LCE] = '1' or HID2[WPE] = '1'. See <i>Section 8</i> on page 229 for a description of this modified behavior.
Data Cache Block Set to Zero Locked	dcbz_l	rA,rB	The dcbz_l instruction is typically a user-level instruction but can be made a supervisor-level instruction.

See *User-Level Cache Instructions (VEA)* on page 105 for cache instructions that provide user-level programs the ability to manage the on-chip caches. If the effective address references a direct-store segment, the instruction is treated as a no-op.

2.3.8 Recommended Simplified Mnemonics

To simplify assembly language coding, a set of alternative mnemonics is provided for some frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Programs written to be portable across the various assemblers for the PowerPC Architecture should not assume the existence of mnemonics not described in this document.

For a complete list of simplified mnemonics, see the simplified Mnemonics section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

3. Espresso Instruction and Data Cache Operation

The Espresso core contains separate 32 KB, 8-way set-associative L1 instruction and data caches to allow the execution units and registers rapid access to instructions and data. This section describes the organization of the on-chip instruction and data caches, the modified, exclusive, shared, invalid (MESI) cache coherency protocol, cache control instructions, various cache operations, and the interaction between the caches, the load/store unit (LSU), the instruction unit, and the memory interface unit (MIU). References to the bus in this section refer to the internal CIU bus, which is shared by the three cores.

The L1 data cache can be configured as a 32 KB normal (hardware-managed) data cache or as a partitioned cache. This section describes the L1 data cache configured as a normal cache, corresponding to `HID2[LCE] = '0'`. When an **mtspr** instruction sets `HID2[LCE] = '1'`, the L1 data cache is partitioned as a 16 KB normal cache and a 16 KB locked cache. The operation of the L1 data cache in this configuration is described in *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229.

There are three Espresso cores on the Espresso chip. In previous designs, the core interfaced to the rest of the system through its BIU to a 60x bus. The Espresso core interfaces to the rest of the system through its memory interface unit (MIU) to the core interface unit (CIU). The CIU implements an internal bus shared among the three cores, which uses a protocol similar to that of the 60x bus to transfer data among cores and memory, and to enforce coherency. The descriptions in this section have maintained some of the terminology associated with the previous design. References to bus transactions and protocol refer to operations on this internal bus. Thus, each of the cores is a bus master and each core snoops the bus for transaction addresses that might be held in its caches. The external 60Xe bus itself is not snooped by Espresso.

The Espresso L1 cache implementation has the following characteristics:

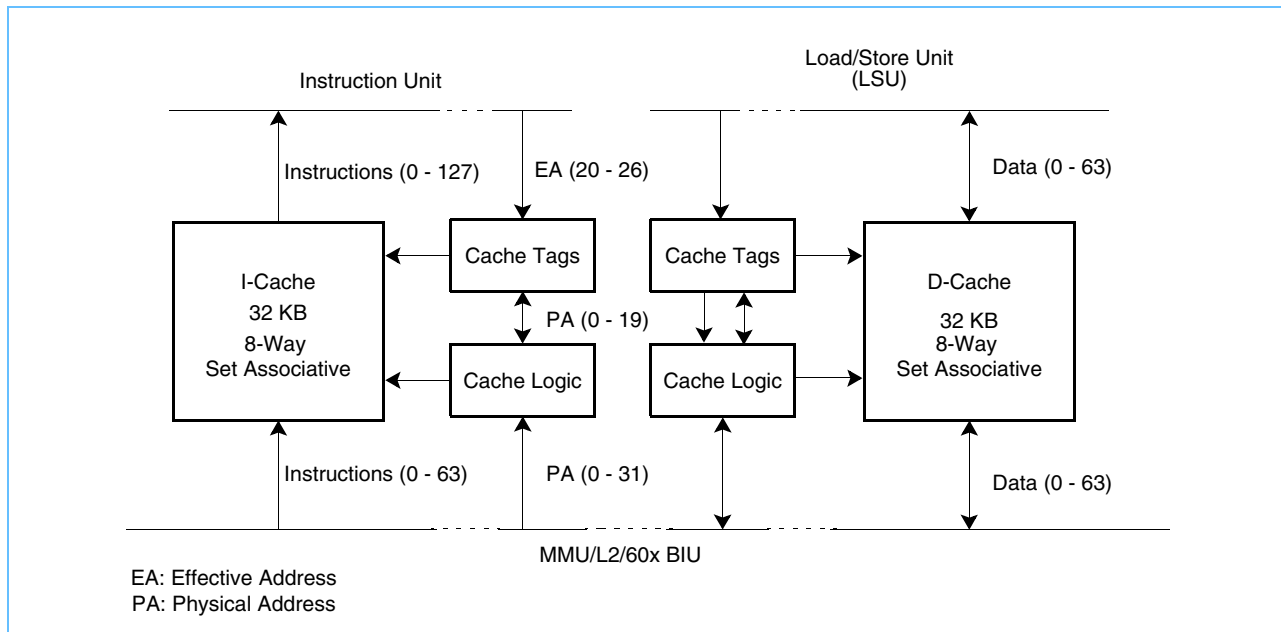
- There are two separate 32 KB instruction and data caches (Harvard architecture).
- Both instruction and data caches are 8-way set associative.
- The caches implement a pseudo least-recently-used (PLRU) replacement algorithm within each set.
- The cache directories are physically addressed. The physical (real) address tag is stored in the cache directory.
- Both the instruction and data caches have 32-byte cache blocks. A cache block is the block of memory that a coherency state describes, also referred to as a cache line.
- Two coherency state bits for each data cache block allow encoding for four states:
 - Modified (Exclusive) (M)
 - Exclusive (Unmodified) (E)
 - Shared (Unmodified) (S)
 - Invalid (I)
- A single coherency state bit for each instruction cache block allows encoding for two possible states:
 - Invalid (INV)
 - Valid (VAL)
- Each cache can be invalidated or locked by setting the appropriate bits in the Hardware Implementation-Dependent Register 0 (HID0), a special-purpose register (SPR) specific to the Espresso core.

Espresso supports a fully-coherent, 4 GB, physical memory address space. The MESI protocol is described in *Section 3.3.2 MESI Protocol* on page 117.

On a cache miss, the Espresso cache blocks are filled in 4 beats of 64 bits each. The burst fill is performed as a critical-doubleword-first operation. The critical doubleword is simultaneously written to the cache and forwarded to the requesting unit, thus minimizing stalls due to cache fill latency. The data cache line is first loaded into a 32-byte reload buffer. When it is full, it is written into the data cache in one cycle. This minimizes the contention between the LSU and the line reload function.

The instruction and data caches are integrated into Espresso as shown in *Figure 3-1*.

Figure 3-1. Cache Integration



Both caches are tightly coupled into the Espresso memory interface unit to allow efficient access to the system memory controller and other cores. The memory interface unit receives requests for bus operations from the instruction and data caches, and forwards the operations to the CIU. The CIU captures snoop addresses for data cache, address queue, and memory reservation (**lwarx** and **stwcx**, instruction) operations. In Espresso, an L1 cache miss first accesses the L2 cache to find the desired cache block before accessing the MIU.

The data cache provides buffers for load and store bus operations. All the data for the corresponding address queues (load and store data queues) is located in the data cache. The data queues are considered temporary storage for the cache and not part of the MIU. The data cache also provides storage for the cache tags required for memory coherency and performs the cache block replacement PLRU function. The data cache is supported by two cache block reload/write-back buffers. This allows a cache block to be loaded or unloaded from the cache in a single cycle.

The data cache supplies data to the general purpose registers (GPRs) and floating-point registers (FPRs) by means of the load/store unit. The Espresso LSU is directly coupled to the data cache to allow efficient movement of data to and from the GPRs and FPRs. The LSU provides all logic required to calculate effective addresses, handles data alignment to and from the data cache, and provides sequencing for load and store string and multiple operations. Write operations to the data cache can be performed on a byte, halfword, word, or doubleword basis.

The instruction cache provides a 128-bit interface to the instruction unit, so that four instructions can be made available to the instruction unit in a single clock cycle. The instruction unit accesses the instruction cache frequently to sustain the high throughput provided by the 6-entry instruction queue.

3.1 Data Cache Organization

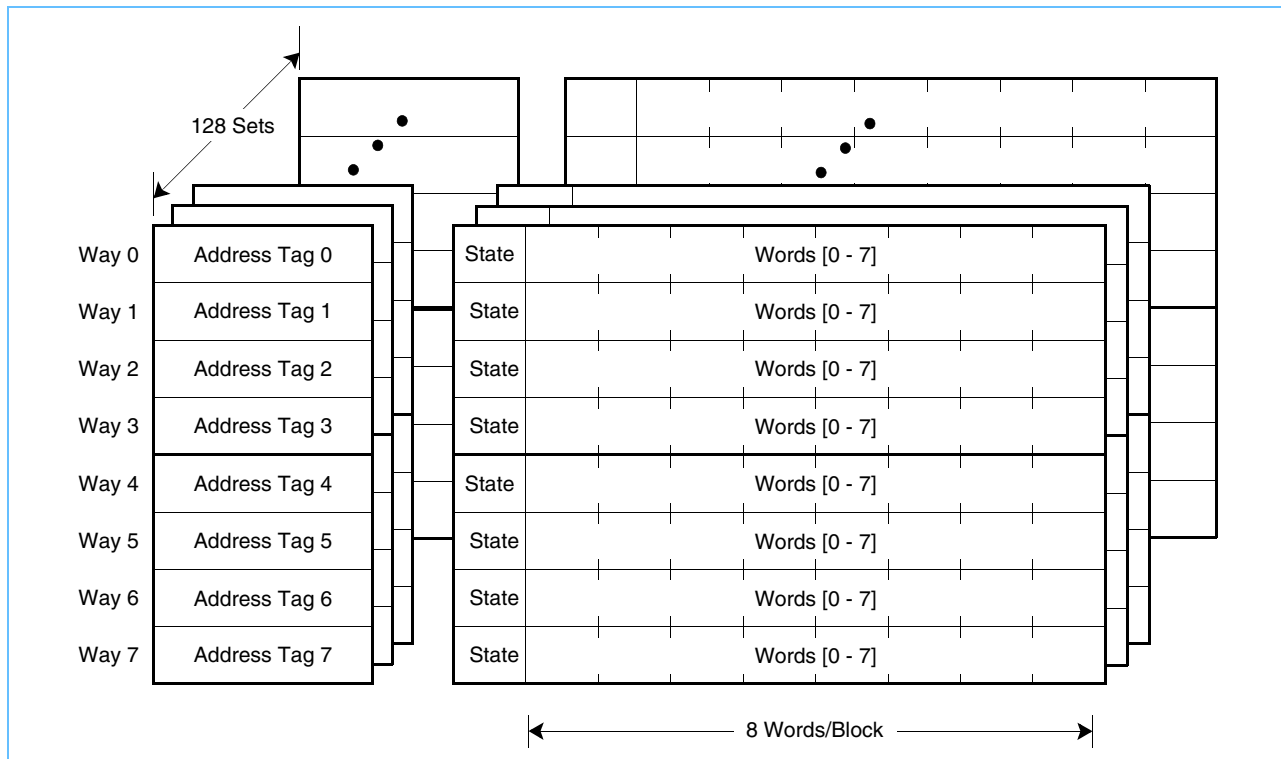
The data cache is organized as 128 sets of eight ways as shown in *Figure 3-2 Data Cache Organization* on page 114. Each way consists of 32 bytes, 2 state bits, and an address tag. Note that in the PowerPC Architecture, the term “cache block,” or simply “block,” when used in the context of cache implementations, refers to the unit of memory at which coherency is maintained. For Espresso, this is the 8-word (32-byte) cache line. This value might be different for other PowerPC implementations.

Each cache block contains eight contiguous words from memory that are loaded from an 8-word boundary (that is, bits A[27:31] of the logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries. Note that address bits A[20:26] provide the index to select a cache set. Bits A[27:31] select a byte within a block. The 2 state bits implement a 4-state MESI (modified/exclusive/shared/invalid) protocol. The MESI protocol is described in *Section 3.3.2 MESI Protocol* on page 117. The tags consist of bits PA[0:19]. Address translation occurs in parallel with set selection (from A[20:26]), and the higher-order address bits (the tag bits in the cache) are physical.

The Espresso on-chip data cache tags are single-ported, and load or store operations must be arbitrated with snoop accesses to the data cache tags. Load or store operations can be performed to the cache on the clock cycle immediately following a snoop access if the snoop misses; snoop hits can block the data cache for two or more cycles, depending on whether a copy-back to main memory is required.

Figure 3-2 shows the Espresso L1 data cache organization.

Figure 3-2. Data Cache Organization



3.2 Instruction Cache Organization

The instruction cache also consists of 128 sets of eight ways, as shown in *Figure 3-3* on page 115. Each way consists of 32 bytes, a single state bit, and an address tag. As with the data cache, each instruction cache block contains 8 contiguous words from memory that are loaded from an 8-word boundary (that is, bits A[27:31] of the logical (effective) addresses are zero); as a result, cache blocks are aligned with page boundaries. Also, address bits A[20:26] provide the index to select a set, and bits A[27:29] select a word within a block.

The tags consist of bits PA[0:19]. Address translation occurs in parallel with set selection (from A[20:26]), and the higher order address bits (the tag bits in the cache) are physical.

The instruction cache differs from the data cache in that it does not implement MESI cache coherency protocol, and a single state bit is implemented that indicates only whether a cache block is valid or invalid. The instruction cache is not snooped. Therefore if a processor modifies a memory location that can be contained in the instruction cache, software must ensure that such memory updates are visible to the instruction fetching mechanism. This can be achieved with the following instruction sequence:

```

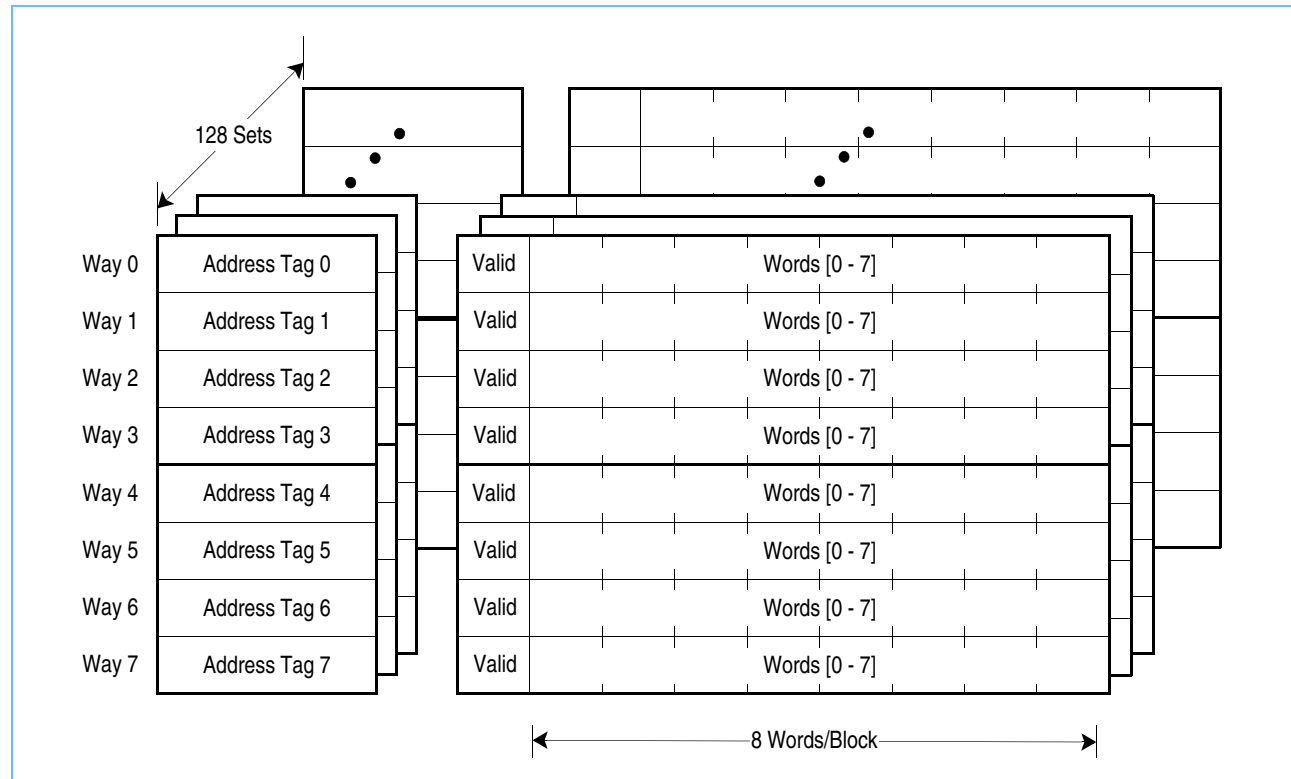
dcbst      # update memory
sync       # wait for update
icbi       # remove (invalidate) copy in instruction cache
sync       # wait for ICBI operation to be globally performed
isync      # remove copy in own instruction buffer

```

These operations are necessary because the processor does not maintain instruction memory coherent with data memory. Software is responsible for enforcing coherency of instruction caches and data memory.

Because instruction fetching can bypass the data cache, changes made to items in the data cache might not be reflected in memory until after the instruction fetch completes.

Figure 3-3. Instruction Cache Organization



3.3 Memory and Cache Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. Coherency allows synchronization and cooperative use of shared resources. Otherwise, multiple copies of a memory location, some containing stale values, can exist in a system resulting in errors when the stale values are used. Each potential bus master must follow rules for managing the state of its cache. This section describes the coherency mechanisms of the PowerPC Architecture and the 4-state cache coherency protocol of the Espresso data cache.

Note: Unless specifically noted, the discussion of coherency in this section applies to the Espresso data cache only. The instruction cache is not snooped, except for **icbi** operations. Instruction cache coherency must be maintained by software. However, Espresso does support a fast instruction cache invalidate capability.

3.3.1 Memory/Cache Access Attributes (WIMG Bits)

Some memory characteristics can be set on either a block by using the WIMG bits in the block address translation (BAT) registers or on a page basis by using the page table entry (PTE), respectively. The WIMG attributes control the following functionality:

- Write-through (W bit)
- Caching-inhibited (I bit)
- Memory coherency (M bit)
- Guarded memory (G bit)

These bits allow both uniprocessor and multiprocessor system designs to exploit numerous system-level performance optimizations.

The WIMG attributes are programmed by the operating system for each page and block. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. The G attribute prevents out-of-order loading and prefetching from the addressed memory location.

The WIMG attributes occupy 4 bits in the BAT registers for block address translation and in the PTEs for page address translation. The WIMG bits are programmed as follows:

- The operating system uses the **mtspr** instruction to program the WIMG bits in the BAT registers for block address translation. The instruction block address translation (IBAT) register pairs do not have a G bit, and all accesses that use the IBAT register pairs are considered not guarded.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The M attribute determines the kind of access performed on the bus (global or local).

Software must exercise care with respect to the use of these bits if coherent memory support is required. Careless specification of these bits can create situations that present coherency paradoxes to the processor. In particular, this can happen when the state of these bits is changed without appropriate precautions (such as flushing the pages that correspond to the changed bits from the caches of all processors in the system) or

when the address translations of aliased real addresses specify different values for any of the WIMG bits. These coherency paradoxes can occur within a single processor or across several processors. It is important to note that, in the presence of a paradox, the operating system software is responsible for correctness.

In real addressing mode, accesses are performed with address translation disabled. (Translation is disabled by setting MSR[IR] = '0' for instructions and MSR[DR] = '0' for data). In real addressing mode, the WIMG bits are automatically generated as '0011' for data and as '0001' for instructions. Thus, the data is write-back, caching is enabled, memory coherency is enforced for data, and memory is guarded

3.3.2 MESI Protocol

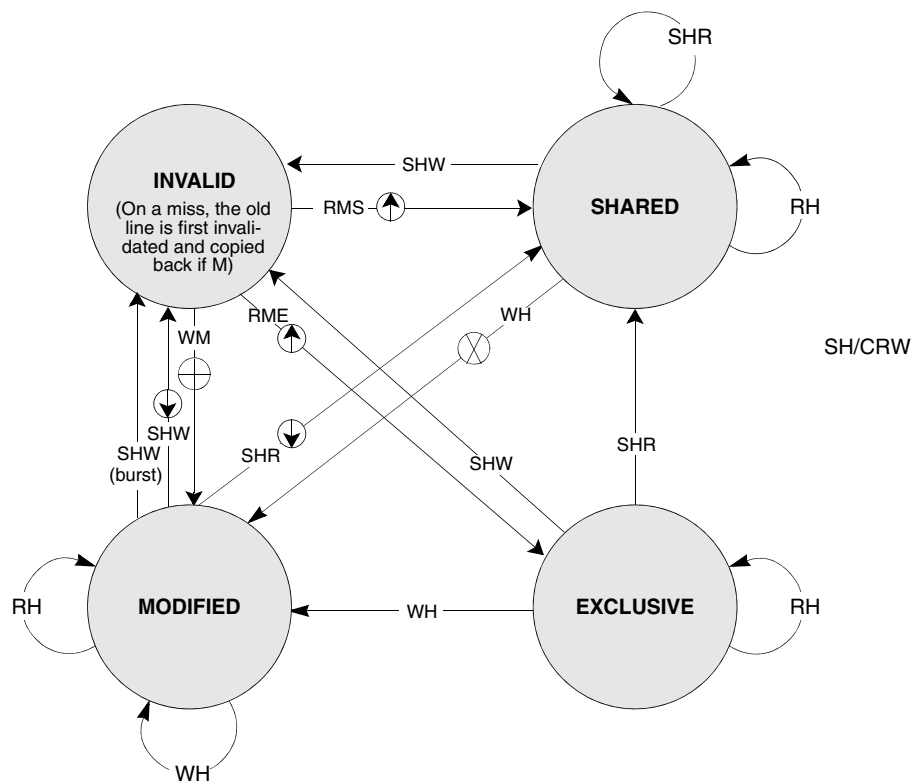
The Espresso data cache coherency protocol is the standard MESI 4-state cache protocol. The Espresso data cache characterizes each 32-byte block it contains as being in one of four MESI states. Addresses presented to the cache are indexed into the cache directory with bits A[20:26], and the upper-order 20 bits from the physical address translation (PA[0:19]) are compared against the indexed cache directory tags. If neither of the indexed tags matches, the result is a cache miss. If a tag matches, a cache hit occurred and the directory indicates the state of the cache block through 2 state bits kept with the tag. The four possible states for a cache block in the cache are the modified state (M), the exclusive state (E), the shared state (S), and the invalid state (I). The four MESI states are defined in *Table 3-1*.

Table 3-1. MESI State Definitions

MESI State	Definition
Modified (M)	The addressed cache block is present in the cache and is modified with respect to system memory; that is, the modified data in the cache block has not been written back to memory. The cache block might be present in the Espresso L2 cache, but it is not present in any other coherent cache.
Exclusive (E)	The addressed cache block is present in the cache, and this cache has exclusive ownership of the addressed block. The addressed block can be present in the Espresso L2 cache, but it is not present in any other processor cache. The data in this cache block is consistent with system memory.
Shared (S)	The addressed block is valid in the cache and might be in at least one other core's cache. This block is always consistent with system memory. That is, the shared state is shared-unmodified.
Invalid (I)	This state indicates that the address block does not contain valid data or that the addressed cache block is not resident in the cache.

Espresso provides dedicated hardware to provide memory coherency by snooping memory transactions. *Figure 3-4 MESI Cache Coherency Protocol State Diagram (WIM = '001')* on page 118 shows the MESI cache coherency protocol as enforced by Espresso. The information in this figure assumes that the WIM bits for the page or block are set to '001'; that is, write-back, caching-not-inhibited, and memory coherency enforced.

Figure 3-4. MESI Cache Coherency Protocol State Diagram (WIM = '001')

**Bus Transactions**

RH = Read Hit
 RMS = Read Miss, Shared
 RME = Read Miss, Exclusive
 WH = Write Hit
 WM = Write Miss
 SHR = Snoop Hit on a Read
 SHW = Snoop Hit on a Write or
 Read-with-Intent-to-Modify

⬇ = Snoop Push

⊗ = Invalidate Transaction

⊕ = Read-with-Intent-to-Modify

⬆ = Cache Block Fill

3.3.2.1 MESI Hardware Considerations

While Espresso provides the hardware required to monitor memory traffic for coherency, the Espresso data cache tags are single-ported. Therefore, a simultaneous load/store and snoop access represents a resource conflict. In general, the snoop access has highest priority and is given first access to the tags. The load or store access occurs on the clock following the snoop. The snoop is not given priority into the tags when the snoop coincides with a tag write (for example, validation after a cache block load). In these situations, the snoop is retried and must re-arbitrate before the lookup is possible.

Occasionally, cache snoops cannot be serviced and must be retried. These retries occur if the cache is busy with a burst read or write when the snoop operation takes place.

Note that it is possible for a snoop to hit a modified cache block that is already in the process of being written to the copy-back buffer for replacement purposes. If this happens, Espresso retries the snoop, and raises the priority of the castout operation to allow it to go to the CIU before the cache block fill.

Another consideration is page table aliasing. If a store hits to a modified cache block but the page table entry is marked write-through (WIMG = 1xxx), then the page has probably been aliased through another page table entry that is marked write-back (WIMG = 0xxx). If this occurs, Espresso ignores the modified bit in the cache tag. The cache block is updated during the write-through operation, and the block remains in the modified state.

The global ($\overline{\text{GBL}}$) signal, asserted as part of the address attribute field during a bus transaction, enables the snooping hardware of Espresso. Address bus masters assert $\overline{\text{GBL}}$ to indicate that the current transaction is a global access (that is, an access to memory shared by more than one device). If $\overline{\text{GBL}}$ is not asserted for the transaction, that transaction is not snooped by Espresso. Note that the $\overline{\text{GBL}}$ signal is not asserted for instruction fetches, and that $\overline{\text{GBL}}$ is asserted for all data read or write operations when using real addressing mode (that is, address translation is disabled).

Normally, $\overline{\text{GBL}}$ reflects the M-bit value specified for the memory reference in the corresponding translation descriptors. Care should be taken to minimize the number of pages marked as global, because the retry protocol enforces coherency and can use considerable bus bandwidth if much data is shared. Therefore, available bus bandwidth decreases as more memory is marked as global.

No snoop update to the Espresso cache occurs if the snooped transaction is not marked global. Also, because cache block castouts and snoop pushes do not require snooping, the $\overline{\text{GBL}}$ signal is not asserted for these operations.

When the Espresso core detects a snoop condition, the associated address is compared with the cache tags. Snooping finishes if no hit is detected. However, if the address hits in the cache, Espresso reacts according to the MESI protocol shown in *Figure 3-4* on page 118.

3.3.3 Coherency Precautions

The following coherency paradoxes can be encountered within a single core:

- Load or store to a caching-inhibited page (WIMG = x1xx) and a cache hit occurs.
The Espresso core ignores any hits to a cache block in a memory space marked caching-inhibited (WIMG = x1xx). The access is performed on the external bus as if there were no hit. The data in the cache is not pushed, and the cache block is not invalidated.
- Store to a page marked write-through (WIMG = 1xxx) and a cache hit occurs to a modified cache block.
The Espresso core ignores the modified bit in the cache tag. The cache block is updated during the write-through operation but the block remains in the modified state (M).

Note that when WIM bits are changed in the page tables or BAT registers, it is critical that the cache contents reflect the new WIM bit settings. For example, if a block or page that had allowed caching becomes caching-inhibited, software should ensure that the appropriate cache blocks are flushed to memory and invalidated.

Also note that aliasing of page translations or of BAT translations in different cores to a given physical address can also lead to these paradoxes. Care must be taken such that all processes running on all cores use the same coherency attributes for any given physical memory address.

3.3.4 Espresso-Initiated Load/Store Operations

Load and store operations are assumed to be weakly ordered on Espresso. The load/store unit (LSU) can perform load operations that occur later in the program ahead of store operations, even when the data cache is disabled. However, strongly ordered load and store operations can be enforced through the setting of the I bit (of the page WIMG bits) when address translation is enabled. Note that when address translation is disabled (real addressing mode), the default WIMG bits cause the I bit to be cleared (accesses are assumed to be cacheable), and thus the accesses are weakly ordered. See *Section 5.2 Real Addressing Mode* on page 170 for a description of the WIMG bits when address translation is disabled.

The Espresso core does not provide support for direct-store segments. Operations attempting to access a direct-store segment invokes a data storage interrupt (DSI) exception. For additional information about DSI exceptions, see *Section 4.5.2 DSI Exception (x'00300')* on page 145.

3.3.4.1 Performed Loads and Stores

The PowerPC Architecture defines a performed load operation as one that has the addressed memory location bound to the target register of the load instruction. The architecture defines a performed store operation as one where the stored value is the value that any other processor receives when executing a load operation (that is, until it is changed again). With respect to Espresso, caching-allowed (WIMG = x0xx) loads and caching-allowed, write-back (WIMG = 00xx) stores are performed when they have arbitrated to address the cache block. Note that in the event of a cache miss, these storage operations can place a memory request into the processor memory queue, but such operations are considered an extension to the state of the cache with respect to snooping bus operations. Caching-inhibited (WIMG = x1xx) loads, caching-inhibited (WIMG = x1xx) stores, and write-through (WIMG = 1xxx) stores are performed when they have been successfully presented to the external 60Xe bus.

3.3.4.2 Sequential Consistency of Memory Accesses

The PowerPC Architecture requires that all memory operations executed by a single processor be sequentially consistent with respect to that processor. This means that all memory accesses appear to be executed in program order with respect to exceptions and data dependencies.

Espresso achieves sequential consistency by operating a single pipeline to the cache and MMU. All memory accesses are presented to the MMU in exact program order, and therefore exceptions are determined in order. Loads are allowed to bypass stores once exception checking has been performed for the store, but data dependency checking is handled in the load/store unit so that a load does not bypass a store with an address match. Note that although memory accesses that miss in the cache are forwarded to the memory queue for future arbitration by the bus, all potential synchronous exceptions have been resolved before the cache. In addition, although subsequent memory accesses can address the cache, full coherency checking between the cache and the memory queue is provided to avoid dependency conflicts.

3.3.4.3 Atomic Memory References

The PowerPC Architecture defines the Load Word and Reserve Indexed (**lwarx**) and the Store Word Conditional Indexed (**stwcx.**) instructions to provide an atomic update function for a single, aligned word of memory. These instructions can be used to develop a rich set of multiprocessor synchronization primitives.

Note: Atomic memory references constructed using **lwarx/stwcx.** instructions depend on the presence of a coherent memory system for correct operation. These instructions should not be expected to provide atomic access to noncoherent memory. For detailed information about these instructions, see *Section 2 Programming Model* on page 49 and *Section 10 Espresso PowerPC Instruction Set* on page 255.

The **lwarx** instruction performs a load word from memory operation and creates a reservation for the 32-byte section of memory that contains the accessed word. The reservation granularity is 32 bytes. The **lwarx** instruction makes a nonspecific reservation with respect to the executing processor and a specific reservation with respect to other masters. This means that any subsequent **stwcx.** executed by the same processor, regardless of address, will cancel the reservation. Also, any bus write or invalidate operation from another processor to an address that matches the reservation address will cancel the reservation.

The **stwcx.** instruction does not check the reservation for a matching address. The **stwcx.** instruction is only required to determine whether a reservation exists. The **stwcx.** instruction performs a store word operation only if the reservation exists. If the reservation has been cancelled for any reason, then the **stwcx.** instruction fails and clears the CR0[EQ] bit in the condition register. The architectural intent is to follow the **lwarx/stwcx.** instruction pair with a conditional branch that checks to see whether the **stwcx.** instruction failed.

If the page table entry is marked caching-allowed (WIMG = x0xx), and an **lwarx** access misses in the cache, then Espresso performs a cache block fill. If the page is marked caching-inhibited (WIMG = x1xx) or the cache is locked and the access misses, then the **lwarx** instruction appears on the bus as a single-beat load. All bus operations that are a direct result of either an **lwarx** instruction or an **stwcx.** instruction are placed on the bus with a special encoding. Note that this does not force all **lwarx** instructions to generate bus transactions, but rather provides a means for identifying when an **lwarx** instruction does generate a bus transaction. If an implementation requires that all **lwarx** instructions generate bus transactions, then the associated pages should be marked as caching-inhibited.

The Espresso data cache treats all **stwcx.** operations as write-through independent of the WIMG settings. When the write-through operation completes successfully, either in the L2 cache or on the 60Xe bus, then the data cache entry is updated (assuming it hits), and CR0[EQ] is modified to reflect the success of the operation. If the reservation is not intact, the **stwcx.** completes in the memory interface unit without performing a

bus transaction, and without modifying either of the caches. As with the **lwarx** instruction, to ensure that the **stwcx.** instruction is correctly broadcast on the bus, the cache line addressed by this instruction should be in a cache-inhibited address space. Alternatively, the **stwcx.** is correctly handled if it is preceded by a **dcbf** instruction to the same address. Also note the restriction on putting a store to the reservation cache line within a **lwarx/stwcx.** loop (see *Section 2.3.4.7 Memory Synchronization Instructions (UISA)* on page 103).

Note: In implementing various synchronization primitives, care must be taken to avoid livelocks. In particular, a scenario that can lead to livelock, or to a lack of parallelism, occurs when one core acquires a lock that controls access to some shared resource, and then attempts to reacquire that lock so soon after releasing it that the other cores do not have a fair chance to acquire the lock. This situation is usually avoided because the core that has just owned the lock has significant work to do associated with that ownership. However, in managing a shared task queue, it is important to avoid any case in which a core can repeatedly acquire and release the lock without significant intervening processing.

3.4 Cache Control

3.4.1 Cache Control Instructions

The PowerPC Architecture defines instructions for controlling both the instruction and data caches (when they exist). The cache control instructions, **dcbt**, **dcbtst**, **dcbz**, **dcbst**, **dcbf**, **dcbi**, and **icbi**, are intended for the management of the local L1 and L2 caches.

The Espresso core broadcasts and snoops all cache control instructions.

The Espresso core implements the **dcbz_I** instruction to allocate lines in the locked cache when $HID2[LCE] = '1'$. See *Section 8 L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe* on page 229 for details.

3.4.1.1 Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst)

The Data Cache Block Touch (**dcbt**) and Data Cache Block Touch for Store (**dcbtst**) instructions provide potential system performance improvement through the use of software-initiated prefetch hints. The Espresso core treats these instructions identically (that is, a **dcbtst** instruction behaves exactly the same as a **dcbt** instruction on the Espresso core).

Note: PowerPC implementations are not required to take any action based on the execution of these instructions, but they can choose to prefetch the cache block corresponding to the effective address into their cache.

The Espresso core loads the data into the cache when the address hits in the translation lookaside buffer (TLB) or the BAT, is permitted load access from the addressed page, is not directed to a direct-store segment, and is directed at a cacheable and unguarded page. Otherwise, Espresso treats these instructions as no-ops. The data brought into the cache as a result of this instruction is validated in the same manner that a load instruction would be (that is, it is marked as exclusive). The memory reference of a **dcbt** (or **dcbtst**) instruction causes the reference bit to be set. Note also that the successful execution of the **dcbt** (or **dcbtst**) instruction affects the state of the TLB and cache least recently used (LRU) bits as defined by the PLRU algorithm.

3.4.1.2 Data Cache Block Zero (*dcbz*)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC Architecture. The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation and protection.

If the block containing the byte addressed by the EA is in the data cache, all bytes are cleared, and the tag is marked as modified (M). If the block containing the byte addressed by the EA is not in the data cache and the corresponding page is caching-allowed, the block is established in the data cache without fetching the block from main memory, all bytes of the block are cleared, and the tag is marked as modified (M).

If the contents of the cache block are from a page marked memory coherence required ($M = 1$), an address-only bus transaction is run before clearing the cache block. The **dcbz** instruction executes regardless of whether the cache is locked, but if the cache is disabled, an alignment exception is generated. If the page containing the byte addressed by the EA is caching-inhibited or write-through, then the system alignment exception handler is invoked. BAT and TLB protection violations generate DSI exceptions.

3.4.1.3 Data Cache Block Store (*dcbst*)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC Architecture. This instruction is treated as a load with respect to address translation and memory protection.

If the address hits in the cache and the cache block is in the exclusive (E) state or shared (S) state, no action is taken. If the address hits in the cache and the cache block is in the modified (M) state, the modified block is written back to memory and the cache block is placed in the exclusive (E) state.

The execution of a **dcbst** instruction broadcasts on the bus. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbst** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.1.4 Data Cache Block Flush (*dcbf*)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC Architecture. This instruction is treated as a load with respect to address translation and memory protection.

If the address hits in the cache and the block is in the modified (M) state, the modified block is written back to memory and the cache block is placed in the invalid (I) state. If the address hits in the cache, and the cache block is in the exclusive (E) state or the shared (S) state, the cache block is placed in the invalid (I) state. If the address misses in the cache, no action is taken.

The execution of **dcbf** broadcasts on the bus. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbf** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.1.5 Data Cache Block Invalidate (*dcbi*)

The effective address is computed, translated, and checked for protection violations as defined in the PowerPC Architecture. This instruction is treated as a store with respect to address translation and memory protection.

If the address hits in the cache, the cache block is placed in the invalid (I) or the shared (S) state, regardless of whether the data is modified. Because this instruction can effectively destroy modified data, it is privileged (that is, **dcbi** is available to programs at the supervisor privilege level, MSR[PR] = '0'). The execution of **dcbi** broadcasts on the bus. The function of this instruction is independent of the WIMG bit settings of the block containing the effective address. The **dcbi** instruction executes regardless of whether the cache is disabled or locked; however, a BAT or TLB protection violation generates a DSI exception.

3.4.1.6 Instruction Cache Block Invalidate (*icbi*)

For the **icbi** instruction, the effective address is not computed or translated, so it cannot generate a protection violation or exception. This instruction performs a virtual lookup into the instruction cache (index only). All ways of the selected instruction cache set are invalidated.

The **icbi** instruction is broadcast on the bus. The **icbi** instruction invalidates the cache blocks independent of whether the cache is disabled or locked.

3.5 Cache Operations

This section describes the Espresso cache operations.

3.5.1 Cache Block Replacement/Castout Operations

Both the instruction and data cache use a pseudo least-recently-used (PLRU) replacement algorithm when a new block needs to be placed in the cache. When the data to be replaced is in the modified (M) state, that data is written into a castout buffer while the missed data is being accessed on the bus. When the load completes, Espresso then pushes the replaced cache block from the castout buffer to the L2 cache (if the L2 cache is enabled) or to the bus (if the L2 is disabled).

The replacement logic first checks to see if there are any invalid blocks in the set and chooses the lowest-order, invalid block (L[0:7]) as the replacement target. If all eight blocks in the set are valid, the PLRU algorithm is used to determine which block should be replaced. The PLRU algorithm is shown in *Figure 3-5 PLRU Replacement Algorithm* on page 125.

Each cache is organized as eight blocks per set by 128 sets. There is a valid bit for each block in the cache, L[0:7]. When all eight blocks in the set are valid, the PLRU algorithm is used to select the replacement target. There are seven PLRU bits, B[0:6], for each set in the cache. For every hit in the cache, the PLRU bits are updated using the rules specified in *Table 3-2. PLRU Bit Update Rules* on page 126.

Figure 3-5. PLRU Replacement Algorithm

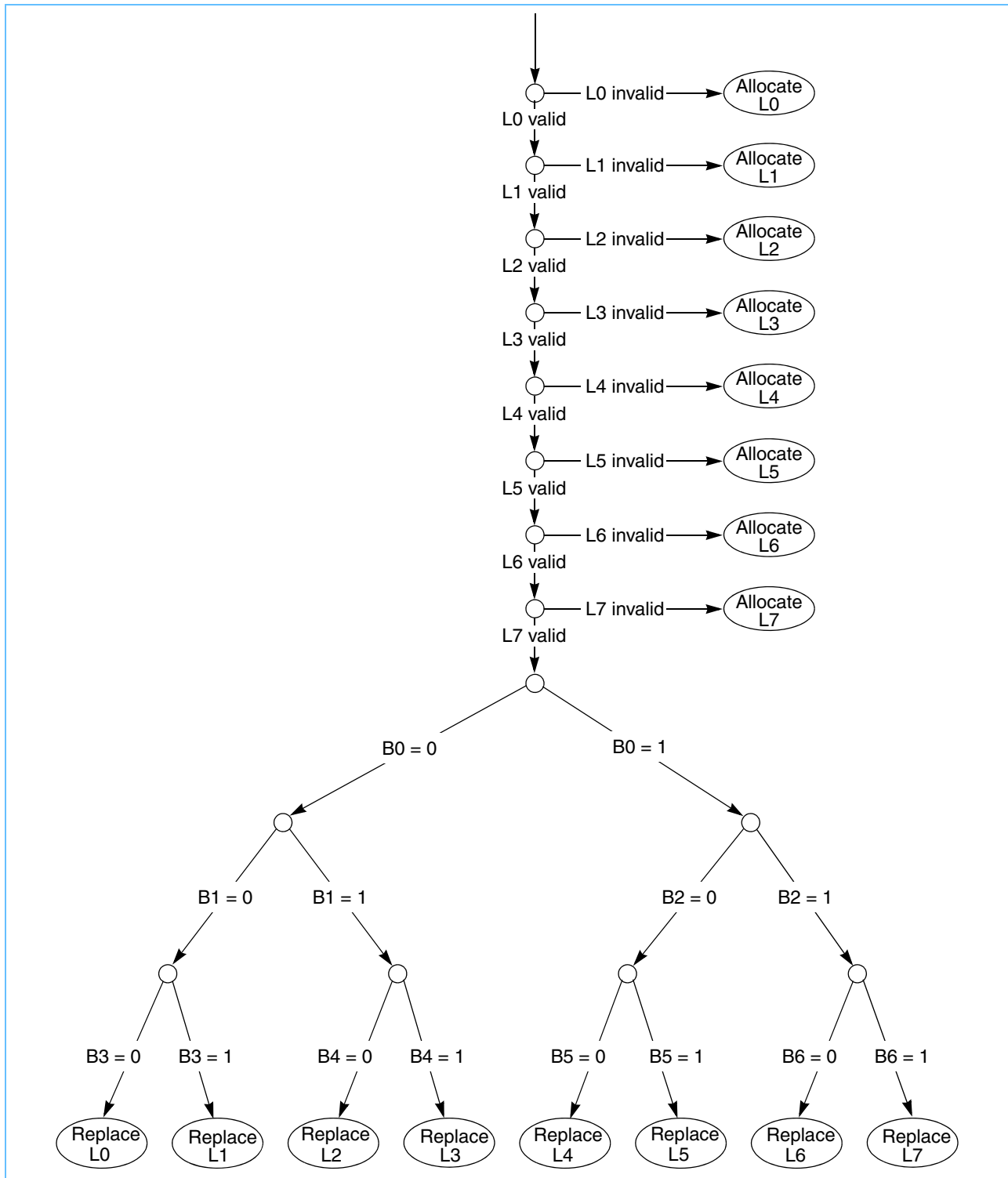


Table 3-2. PLRU Bit Update Rules

If the Current Access is to:	Then the PLRU Bits are changed to: ¹						
	B0	B1	B2	B3	B4	B5	B6
L0	1	1	x	1	x	x	x
L1	1	1	x	0	x	x	x
L2	1	0	x	x	1	x	x
L3	1	0	x	x	0	x	x
L4	0	x	1	x	x	1	x
L5	0	x	1	x	x	0	x
L6	0	x	0	x	x	x	1
L7	0	x	0	x	x	x	0

1. x = Does not change.

If all eight blocks are valid, then a block is selected for replacement according to the PLRU bit encodings shown in *Table 3-3*.

Table 3-3. PLRU Replacement Block Selection

If the PLRU Bits are:						Then the Block Selected for Replacement is:
B0	0	B1	0	B3	0	L0
	0		0		1	L1
	0		1	B4	0	L2
	0		1		1	L3
	1	B2	0	B5	0	L4
	1		0		1	L5
	1		1	B6	0	L6
	1		1		1	L7

During power-up or hard reset, all the valid bits of the blocks are cleared and the PLRU bits are cleared to point to block L0 of each set. Note that this is also the state of the data or instruction cache after setting their respective flash invalidate bit (HID0[DCF_I] or HID0[ICF_I]).

When the L1 cache is partitioned into a locked 16 KB array and a normal 16 KB array, the PLRU replacement algorithm corresponds to the extension of the unpartitioned cache. For other than **dcbz_l** accesses, the following occurs:

- If way 0 is not valid, it is selected for replacement.
- Otherwise, if way 1 is not valid, it is selected for replacement.
- Otherwise, if way 2 is not valid, it is selected for replacement.
- Otherwise, if way 3 is not valid, it is selected for replacement.
- Otherwise, replacement is as described in *Table 3-4* on page 127.

Similarly, for **dcbz_I** accesses, the following occurs:

- If way 4 is not valid, it is selected for replacement.
- Otherwise, if way 5 is not valid, it is selected for replacement.
- Otherwise, if way 6 is not valid, it is selected for replacement.
- Otherwise, if way 7 is not valid, it is selected for replacement.
- Otherwise, replacement is as described in *Table 3-4* on page 127.

Table 3-4. Partitioned L1 PLRU Replacement Algorithm

If the PLRU bits are:					The way replaced is:
non dcbz_I access	B1	0	B3	0	0
		0		1	1
		1	B4	0	2
		1		1	3
dcbz_I access	B2	0	B5	0	4
		0		1	5
		1	B6	0	6
		1		1	7

3.5.2 Cache Flush Operations

The instruction cache can be invalidated by executing a series of **icbi** instructions or by setting **HID0[ICFI]**. The data cache can be invalidated by executing a series of **dcbi** instructions or by setting **HID0[DCF]**.

Any modified entries in the data cache can be copied back to memory (flushed) by using the **dcbf** instruction or by executing a series of 12 uniquely addressed load or **dcbz** instructions to each of the 128 sets. The replacement algorithm described in *Figure 5-3 Espresso Microprocessor DMMU Block Diagram* on page 157 requires 12 loads or stores to guarantee that all eight lines in each set will be replaced. The address space should not be shared with any other process to prevent snoop hit invalidations during the flushing routine. Exceptions should be disabled during this time so that the PLRU algorithm is not disturbed.

Under the same conditions as described previously (using a nonshared address space and with exceptions disabled), the data cache flush assist bit, **HID0[DCFA]**, can be used to simplify the software flushing process. When set, **HID0[DCFA]** forces the PLRU replacement algorithm to ignore the invalid entries and follow the replacement sequence defined by the PLRU bits. This reduces the series of uniquely addressed load or **dcbz** instructions to eight per set. **HID0[DCFA]** should be set just before the beginning of the cache flush routine and cleared after the series of instructions has completed.

3.5.3 Data Cache-Block-Fill Operations

The core data cache block reload buffer is filled in 4 beats of 64 bits each, with the critical doubleword loaded first; the data is then written to the cache in one cycle. The data cache is not blocked to internal accesses while the load (caused by a cache miss) completes. This functionality is sometimes referred to as “hits under misses,” because the cache can service a hit while a cache miss fill is waiting to complete. The critical-doubleword read from memory is simultaneously written to the data cache reload buffer and forwarded to the requesting unit, thus minimizing stalls due to cache fill latency.

A cache block is filled after a read miss or write miss (read-with-intent-to-modify) occurs in the cache. The cache block that corresponds to the missed address is updated by a burst transfer of the data from the L2 cache or system memory. Note that if a read miss occurs in a system with multiple bus masters, and the data is modified in another cache, the modified data is pushed to the bus before the cache fill occurs.

3.5.4 Instruction Cache-Block-Fill Operations

The Espresso instruction cache blocks are loaded in 4 beats of 64 bits each, with the critical doubleword loaded first. The instruction cache is not blocked to internal accesses while the fetch (caused by a cache miss) completes. On a cache miss, the critical and following doublewords read from memory are simultaneously written to the instruction cache and forwarded to the instruction queue, thus minimizing stalls due to cache fill latency. There is no snooping of the instruction cache except for **icbi** operations.

3.5.5 Data Cache-Block-Push Operation

When a cache block in the Espresso core is snooped and hit by another bus master and the data is modified, the cache block must be pushed to the bus and made available to the snooping device. Espresso supports normal push operations.

3.6 L1 Caches and Bus Transactions

The Espresso core transfers data to and from the cache in single-beat transactions of 2 words, or in 4-beat transactions of 8 words that fill a cache block. Single-beat bus transactions can transfer from 1 - 8 bytes to or from Espresso, and can be misaligned.

Burst transactions on Espresso always transfer 8 words of data at a time, and are aligned to a doubleword boundary. Burst transactions have an assumed address order. For cacheable read operations, instruction fetches, or cacheable, non-write-through write operations that miss the cache, the Espresso core presents the doubleword-aligned address associated with the load/store instruction or instruction fetch that initiated the transaction.

The first quadword contains data from the address of the load/store or instruction fetch that missed the cache. This minimizes latency by allowing the critical code or data to be forwarded to the processor before the rest of the block is filled. For all other burst operations, however, the entire block is transferred in order (octword aligned). Critical-doubleword-first fetching on a cache miss applies to both the data and instruction cache.

3.6.1 Snooping

The Espresso core maintains data cache coherency in hardware by coordinating activity between the data cache, the memory interface logic, the L2 cache, and the memory system. Espresso has a copy-back cache that relies on bus snooping to maintain cache coherency with other caches in the system. For Espresso, the coherency size of the bus is the size of a cache block: 32 bytes. This means that any bus transactions that cross an aligned 32-byte boundary must present a new address onto the bus at that boundary for correct snoop operation by Espresso, or they must operate noncoherently with respect to Espresso.

As bus operations are performed on the bus by other bus masters, the Espresso bus snooping logic monitors the addresses and transfer attributes that are referenced. The Espresso core snoops the bus transactions for any of the following snoop conditions:

- The global signal is asserted indicating that coherency enforcement is required.

- A reservation is currently active in the Espresso core as the result of an **lwarx** instruction, and the transfer type attributes indicate a write or kill operation. These transactions are snooped regardless of whether global is asserted to support reservations in the MESI cache protocol.

Once a snoop condition is detected on the bus, the snooped address is compared against the data cache tags, memory queues, and/or other storage elements as appropriate except for **icbi** operations. The L1 data cache tags and L2 cache tags are snooped for standard data cache coherency support. No snooping is done in the instruction cache for coherency.

The memory queues are snooped for pipeline collisions and memory coherency collisions. A pipeline collision is detected when another bus master addresses any portion of a line that this Espresso data cache is currently in the process of loading (L1 loading from L2, or L1 or L2 loading from the bus). A memory coherency collision occurs when another bus master addresses any portion of a line that Espresso has currently queued to write to memory from the data cache (castout or copy-back), but has not yet been granted bus access to perform.

If a snooped transaction results in a cache hit or pipeline collision or memory queue collision, the Espresso core asserts address retry on the bus. The current bus master, detecting the assertion of address retry, aborts the transaction and retries it at a later time, so that the Espresso snooped core can first perform a write operation back to memory from its cache or memory queues. Espresso can also retry a bus transaction if it is unable to snoop the transaction on that cycle due to internal resource conflicts. Additional snoop action can be forwarded to the cache as a result of a snoop hit in some cases (a cache push of modified data, or a cache block invalidation). There is no immediate way for another CPU bus agent to determine the cause of the Espresso address retry.

Implementation Note: Snooping of the memory queues for pipeline collisions, as described previously, is performed for burst read operations in progress only. In this case, the read address has completed on the bus; however, the data tenure can be either in-progress or not yet started by the processor. During this time, Espresso retries any other global access to that line by another bus master until all data has been received in its L1 cache. Pipeline collisions, however, do not apply for burst write operations in progress. If Espresso has completed an address tenure for a burst write and is currently waiting for a data bus grant or is currently transferring data to memory, it does not generate an address retry to another bus master that addresses the line. The CIU handles this in the Espresso chip by keeping the data transactions to memory in order. Note also that all burst writes by Espresso are performed as nonglobal, and hence do not typically enable snooping even for address collision purposes. (Snooping can still occur for reservation cancelling purposes.) Operations associated with data cache block instructions are not snooped in these queues. This can occasionally result in an unnecessary CLEAN operation being broadcast to an address that has recently been snooped, but has no other ramifications.

3.7 MESI State Transactions

Table 3-5 shows MESI state transitions for various operations.

Table 3-5. L1 Data Cache State Transitions in Response to LSU Operations (Page 1 of 2)

Instruction Type	WT	CI	Current State	Next State	Action	Transaction Type
Load	-	0	I	E,S,M	Castout victim	Write-with-kill
					Request from memory	Read
			E, S, M	Same	Read from cache	-
		1	I, E, S	Same	Request from memory	Single-beat read
			M*	Same	Request from memory	Single-beat read
lwarx	-	0	I	E,S,M	Castout victim	Write-with-kill
					Request from memory	ReadA
			E, S, M	Same	Read from cache	-
		1	I, E, S	Same	Request from memory	Single-beat readA
			M*	Same	Request from memory	Single-beat readA
Store	0	0	I	M	Castout victim	Write-with-kill
					Request from memory	RWITM
					Write to cache	-
			E, M	M	Write to cache	-
			S	M	Request from memory	RWITM
					Write to cache	-
		1	I	Same	Write to memory	Single-beat Write-with-flush
			E*	Same	Write to memory	Single-beat Write-with-flush
			S*	Same	Write to memory	Single-beat Write-with-flush
			M*	Same	Write to memory	Single-beat Write-with-flush
	1	0	I	Same	Write to memory	Single-beat Write-with-flush
					Write to cache	-
			E	Same	Write to memory	Single-beat Write-with-flush
					Write to cache	-
			S	E	Write to memory	Single-beat Write-with-flush
					Write to cache	-
			M*	Same	Write to memory	Single-beat Write-with-flush

Table 3-5. L1 Data Cache State Transitions in Response to LSU Operations (Page 2 of 2)

Instruction Type	WT	CI	Current State	Next State	Action	Transaction Type
stwcx. with reservation set	–	–	I	Same	Write to memory	Single-beat Write-with-flush atomic
			E	Same	Write to cache	
					Write to memory	Single-beat Write-with-flush atomic
			S	E	Write to cache	–
					Write to memory	Single-beat Write-with-flush atomic
			M	Same	Write to cache	–
					Write to memory	Single-beat Write-with-flush atomic
dcbf	–	–	I, E, S	I	Flush other copies	Flush
			M	I	Invalidate L2 Push line to memory	Write-with-kill
dcbst	–	–	I, E, S	Same	Clean other copies	Clean
			M	E	Invalidate L2 Push line to memory	Write-with-kill
dcbi	–	–	I, E, S, M	I	Kill other copies	Kill
dcbz	0	0	I	M	Castout victim	Write-with-kill
					Kill other copies	Kill
					Write zeros to cache	–
			E, M	M	Write zeros to cache	–
					Kill other copies	Kill
					Write zeros to cache	–
	1	–	–	–	Alignment exception	–
	–	1	–	–	Alignment exception	–
dcbz_l	0	0	I, E, S, M	M	Write zeros to cache	–
	1	–	–	–	Alignment exception	–
	–	1	–	–	Alignment exception	–
dcbt dcbtst	–	0	I	E, S, M	Castout victim	Write-with-kill
					Read from memory	Read
		1	E, S, M	Same	–	–
					–	–



4. Exceptions

The OEA portion of the PowerPC Architecture defines the mechanism by which PowerPC processors implement exceptions (referred to as interrupts in the architecture specification). Exception conditions can be defined at other levels of the architecture. For example, the UISA defines conditions that can cause floating-point exceptions; the OEA defines the mechanism by which the exception is taken.

The PowerPC exception mechanism allows the processor to change to supervisor state as a result of unusual conditions arising in the execution of instructions and from external signals, bus errors, or various internal conditions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (exception vector) predetermined for each exception. Processing of exceptions begins in supervisor mode.

Although multiple exception conditions can map to a single exception vector, often a more specific condition can be determined by examining a register associated with the exception (for example, the Data Storage Interrupt Status Register [DSISR] and the Floating-Point Status and Control Register [FPSCR]). The high-order bits of the MSR are also set for some exceptions. Also, software can explicitly enable or disable some exception conditions.

The PowerPC Architecture requires that exceptions be taken in program order; therefore, although a particular implementation might recognize exception conditions out of order, they are handled strictly in order with respect to the instruction stream. When an instruction-caused exception is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the exception is taken. For example, if a single instruction encounters multiple exception conditions, those exceptions are taken and handled based on the priority of the exception. Likewise, exceptions that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

To prevent loss of state information, exception handlers must save the information stored in the Machine Status Save/Restore Registers, SRR0 and SRR1, soon after the exception is taken to prevent this information from being lost due to another exception being taken. Because exceptions can occur while an exception handler routine is executing, multiple exceptions can become nested. It is up to the exception handler to save the necessary state information if control is to return to the excepting program.

In many cases, after the exception handler returns, there is an attempt to execute the instruction that caused the exception (for example, page fault). Instruction execution continues until the next exception condition is encountered. Recognizing and handling exception conditions sequentially guarantees that the machine state is recoverable and that processing can resume without losing instruction results.

In this book, the following terms are used to describe the stages of exception processing:

Recognition	Exception recognition occurs when the condition that can cause an exception is identified by the processor.
Taken	An exception is said to be taken when control of instruction execution is passed to the exception handler; that is, the context is saved, the instruction at the appropriate vector offset is fetched, and the exception handler routine is begun in supervisor mode.
Handling	Exception handling is performed by the software linked to the appropriate vector offset. Exception handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

Note: The PowerPC Architecture documentation refers to exceptions as interrupts. In this book, the term “interrupt” is reserved to refer to asynchronous exceptions and sometimes to the event that causes the exception. Also, the PowerPC Architecture uses the word “exception” to refer to IEEE-defined floating-point exception conditions that can cause a program exception to be taken; see *Section 4.5 Exception Definitions* on page 143. The occurrence of these IEEE exceptions might not cause an exception to be taken. IEEE-defined exceptions are referred to as IEEE floating-point exceptions or floating-point exceptions.

4.1 Microprocessor Exceptions

As specified by the PowerPC Architecture, exceptions can be either precise or imprecise and either synchronous or asynchronous. Asynchronous exceptions are caused by events external to the processor execution; synchronous exceptions are caused by instructions.

The types of exceptions are shown in *Table 4-1*.

Note: All exceptions except for the performance monitor exception are defined, at least to some extent, by the PowerPC Architecture.

Table 4-1. Espresso Microprocessor Exception Classifications

Synchronous/Asynchronous	Precise/Imprecise	Exception Types
Asynchronous, nonmaskable	Imprecise	Machine check, system reset
Asynchronous, maskable	Precise	External interrupt, decremter, and performance monitor interrupt
Synchronous	Precise	Instruction-caused exceptions

These classifications are discussed in greater detail in *Section 4.2 Exception Recognition and Priorities* on page 135.

For a better understanding of how the Espresso microprocessor implements precise exceptions, see the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual. Exceptions implemented in Espresso, and conditions that cause them, are listed in *Table 4-2*.

Table 4-2. Exceptions and Conditions (Page 1 of 2)

Exception Type	Vector Offset (hexadecimal)	Causing Conditions
Reserved	00000	—
System reset	00100	Assertion of either $\overline{\text{HRESET}}$ or $\overline{\text{SRESET_Cx}}$ or at power-on reset.
Machine check	00200	Assertion of $\overline{\text{MCP_Cx}}$, an address or data error, DMA queue overflow, DMA look-up misses locked cache, or dcbz_I cache hit. MSR[ME] must be set.
Data storage interrupt	00300	As specified in the PowerPC Architecture. For translation lookaside buffer (TLB) misses on load, store, or cache operations, a DSI exception occurs if a page fault occurs.
Instruction storage interrupt	00400	As defined by the PowerPC Architecture.
External interrupt	00500	MSR[EE] = '1' and $\overline{\text{INT_Cx}}$ is asserted.

Table 4-2. Exceptions and Conditions (Page 2 of 2)

Exception Type	Vector Offset (hexadecimal)	Causing Conditions
Alignment	00600	- A floating-point load/store, stmw, stwcx., lmw, lwarx, eciw, or ecow instruction operand is not word-aligned. - A multiple/string load/store operation is attempted in little-endian mode. - An operand of a dcbz or dcbz_l instruction is on a page that is write-through or cache-inhibited for a virtual mode access. - An attempt to execute a dcbz or dcbz_l instruction occurs when the cache is disabled.
Program	00700	As defined by the PowerPC Architecture.
Floating-point unavailable	00800	As defined by the PowerPC Architecture.
Decrementer	00900	As defined by the PowerPC Architecture, when the most-significant bit of the DEC Register changes from '0' to '1' and MSR[EE] = '1'.
Reserved	00A00 - 00BFF	—
System call	00C00	Execution of the System Call (sc) instruction.
Trace	00D00	MSR[SE] = '1' or a branch instruction is completing and MSR[BE] = '1'. The Espresso core differs from the OEA by not taking this exception on an isync .
Reserved	00E00	The Espresso core does not generate an exception to this vector. Other PowerPC processors might use this vector for floating-point assist exceptions.
Reserved	00E10 - 00EFF	—
Performance monitor	00F00	The limit specified in PMC <i>n</i> is met and MMCR0[ENINT] = '1' (Espresso-specific).
Instruction address breakpoint	01300	IABR[0:29] matches EA[0 - 29] of the next instruction to complete, IABR[TE] matches MSR[IR], and IABR[BE] = '1'. (Espresso-specific)
Reserved	01400 - 02FFF	—

4.2 Exception Recognition and Priorities

Exceptions are roughly prioritized by exception class, as follows:

1. Nonmaskable, asynchronous exceptions have priority over all other exceptions. The system reset and machine check exceptions cannot be delayed and do not wait for completion of any precise exception handling, although the machine check exception condition can be disabled so that the condition causes the processor to go directly into the checkstop state.
2. Synchronous, precise exceptions are caused by instructions and are taken in strict program order.
3. Imprecise exceptions (imprecise mode floating-point enabled exceptions) are caused by instructions and they are delayed until higher priority exceptions are taken. Note that the Espresso microprocessor does not implement an exception of this type.
4. Maskable asynchronous exceptions (external, decrementer, and performance monitor exceptions) are delayed if higher priority exceptions are taken.

The following list of exception categories describes how Espresso handles exceptions up to the point of signaling the appropriate interrupt to occur. Note that a recoverable state is reached if the completed store queue is empty (drained, not cancelled) and any instruction that is next in program order and has been

signaled to complete has completed. If MSR[RI] = '0', Espresso is in a nonrecoverable state. Also, instruction completion is defined as updating all architectural registers associated with that instruction and then removing that instruction from the completion buffer.

- Exceptions caused by asynchronous events (interrupts). These exceptions are distinguished by whether they are maskable and recoverable.
 - Asynchronous, nonmaskable, nonrecoverable
System reset for assertion of $\overline{\text{HRESET}}$. Has highest priority and is taken immediately regardless of other pending exceptions or recoverability. (Includes power-on reset)
 - Asynchronous, maskable, nonrecoverable
Machine check exception. Has priority over any other pending exception except system reset for assertion of $\overline{\text{HRESET}}$. Taken immediately regardless of recoverability.
 - Asynchronous, nonmaskable, recoverable
System reset for $\overline{\text{SRESET}}$. Has priority over any other pending exception except system reset for $\overline{\text{HRESET}}$ (or power-on reset), or machine check. Taken immediately when a recoverable state is reached.
 - Asynchronous, maskable, recoverable
Performance monitor, external, and decremter interrupts. Before handling this type of exception, the next instruction in program order must complete. If that instruction causes another type of exception, that exception is taken, and the asynchronous, maskable recoverable exception remains pending until the instruction completes. Further instruction completion is halted. The asynchronous, maskable recoverable exception is taken when a recoverable state is reached.
- Instruction-related exceptions. These exceptions are further organized into the point in instruction processing in which they generate an exception.
 - Instruction fetch
ISI exceptions. Once this type of exception is detected, dispatching stops and the current instruction stream is allowed to drain out of the machine. If completing any of the instructions in this stream causes an exception, that exception is taken and the instruction fetch exception is discarded (but might be encountered again when instruction processing resumes). Otherwise, once all pending instructions have executed and a recoverable state is reached, the ISI exception is taken.
 - Instruction dispatch/execution
Program, DSI, alignment, floating-point unavailable, system call, and instruction address breakpoint. This type of exception is determined during dispatch or execution of an instruction. The exception remains pending until all instructions before the exception-causing instruction in program order complete. The exception is then taken without completing the exception-causing instruction. If completing these previous instructions causes an exception, that exception takes priority over the pending instruction dispatch/execution exception, which is then discarded (but might be encountered again when instruction processing resumes).
 - Post-instruction execution
Trace. Trace exceptions are generated following execution and completion of an instruction while trace mode is enabled. If executing the instruction produces conditions for another type of exception, that exception is taken and the post-instruction exception is forgotten for that instruction.

Note: These exception classifications correspond to how exceptions are prioritized, as described in *Table 4-3* on page 137.

Table 4-3. Espresso Exception Priorities

Priority	Exception	Cause
Asynchronous Exceptions (Interrupts)		
0	System reset	Power-on reset, assertion of $\overline{\text{HRESET}}$ and $\overline{\text{TRST}}$ (hard reset).
1	Machine check	Any enabled machine check condition (assertion of $\overline{\text{MCP_Cx}}$).
2	System reset	Assertion of soft reset ($\overline{\text{SRESET_Cx}}$).
3	External interrupt	Assertion of interrupt ($\overline{\text{INT_Cx}}$).
4	Performance monitor	Any programmer-specified performance monitor condition.
5	Decrementer	Decrementer passes through zero.
Instruction Fetch Exceptions		
0	ISI	Any ISI exception condition.
Instruction Dispatch/Execution Exceptions		
0	Instruction address breakpoint	Any instruction address breakpoint exception condition.
1	Program	Occurrence of an illegal instruction, privileged instruction, or trap exception condition. Note that floating-point enabled program exceptions have lower priority.
2	System call	System call (sc) instruction.
3	Floating-point unavailable	Any floating-point unavailable exception condition.
4	Program	A floating-point enabled exception condition (lowest-priority program exception).
5	DSI	DSI exception due to eciwx , ecowx with $\text{EAR}[E] = '0'$ ($\text{DSISR}[11]$). Lower priority DSI exception conditions are shown in priorities 7 - 11.
6	Alignment	Any alignment exception condition, prioritized as follows: 1 Floating-point access not word-aligned. 2 lmw , stmw , lwarx , stwcx . not word-aligned. 3 eciwx or ecowx not word-aligned. 4 Multiple or string access with $\text{MSR}[\text{LE}]$ set. 5 dcbz or dcbz_l to write-through or cache-inhibited page or cache is disabled.
7	DSI	Page fault.
8	DSI	Block address translation (BAT) page protection violation.
9	DSI	Any access except cache operations to a segment where $\text{SR}[\text{T}] = '1'$ ($\text{DSISR}[5]$) or an access crosses from a $\text{T} = '0'$ segment to one where $\text{T} = '1'$ ($\text{DSISR}[5]$).
10	DSI	TLB page protection violation.
11	DSI	DABR address match.
Post-Instruction Execution Exceptions		
12	Trace	$\text{MSR}[\text{SE}] = '1'$ (or $\text{MSR}[\text{BE}] = '1'$ for branches).

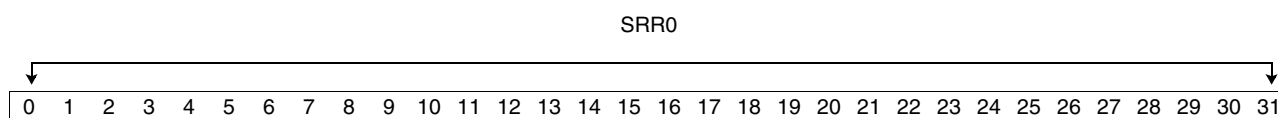
System reset and machine check exceptions can occur at any time and are not delayed even if an exception is being handled. As a result, state information for an interrupted exception might be lost; therefore, these exceptions are typically nonrecoverable. An exception might not be taken immediately when it is recognized.

4.3 Exception Processing

When an exception is taken, the processor uses SRR0 and SRR1 to save the contents of the MSR for the current context and to identify where instruction execution should resume after the exception is handled.

4.3.1 Machine Status Save/Restore Register 0 (SRR0)

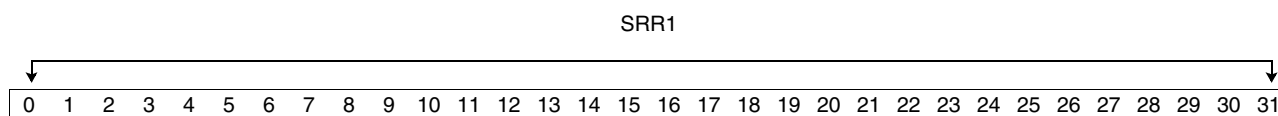
When an exception occurs, the address saved in SRR0 determines where instruction processing should resume when the exception handler returns control to the interrupted process. Depending on the exception, this might be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This might be the address of the instruction that caused the exception or the next one (as in the case of a system call, trace, or trap exception).



Bits	Field Name	Description
0:31	SRR0	Holds the effective address (EA) for the instruction in the interrupted program flow.

4.3.2 Machine Status Save/Restore Register 1 (SRR1)

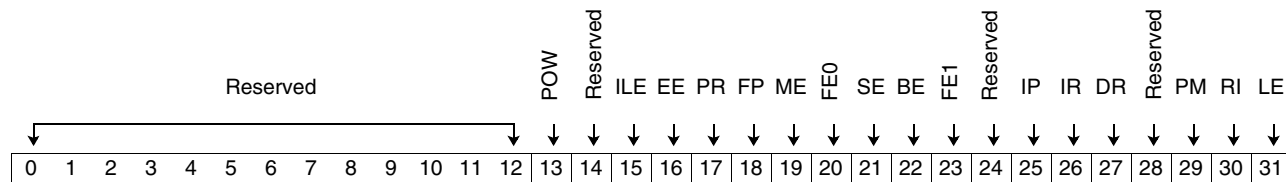
SRR1 is used to save machine status (selected MSR bits and possibly other status bits as well) on exceptions and to restore those values when an **rfi** instruction is executed.



Bits	Field Name	Description
0:31	SRR1	Exception-specific information and MSR bit values

4.3.3 Machine State Register (MSR)

For most exceptions, SRR1[2:4,10:12] are loaded with exception-specific information and MSR[5:9, 16:31] are placed into the corresponding bit positions of SRR1.



Bits	Field Name	Description
0	—	Reserved. Full function. ¹
1:4	—	Reserved. Partial function. ¹
5:9	—	Reserved. Full function. ¹
10:12	—	Reserved. Partial function. ¹
13	POW	Power management enable. 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode). Power management functions are implementation-dependent.
14	—	Reserved. Partial function. ¹
15	ILE	Exception little-endian mode. When an exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception.
16	EE	External interrupt enable. 0 The processor delays recognition of external interrupts and decremter exception conditions. 1 The processor is enabled to take an external interrupt or the decremter exception.
17	PR	Privilege level. 0 The processor can execute both user-level and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available. 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions and can take floating-point enabled program exceptions.
19	ME	Machine check enable. 0 Machine check exceptions are disabled. If one occurs, the system enters checkstop. 1 Machine check exceptions are enabled.
20	FE0	IEEE floating-point exception mode 0 (see <i>Table 4-4</i> on page 140).
21	SE	Single-step trace enable. 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of every instruction except rfi , isync , and sc . Successful execution means that the instruction caused no other exception.

1. Full function reserved bits are saved in the SRR1 when an exception occurs; partial function reserved bits are not saved.

Bits	Field Name	Description
22	BE	Branch trace enable. 0 The processor executes branch instructions normally. 1 The processor generates a branch type trace exception when a branch instruction executes successfully.
23	FE1	IEEE floating-point exception mode 1 (see <i>Table 4-4</i> on page 140).
24	—	Reserved. This bit corresponds to the AL bit of the Power Architecture.
25	IP	Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, <i>nnnn</i> is the offset of the exception. 0 Exceptions are vectored to the physical address x'000n_nnnn'. 1 Exceptions are vectored to the physical address x'FFFn_nnnn'.
26	IR	Instruction address translation. 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information, see <i>Section 5 Memory Management</i> on page 151.
27	DR	Data address translation. 0 Data address translation is disabled. 1 Data address translation is enabled. For more information, see <i>Section 5</i> on page 151.
28	—	Reserved. Full function. ¹
29	PM	Performance monitor marked mode. 0 Process is not a marked process. 1 Process is a marked process. Espresso specific; defined as reserved by the PowerPC Architecture. For more information about the performance monitor, see <i>Section 4.5.12</i> on page 149.
30	RI	Indicates whether a system reset or machine check exception is recoverable. 0 Exception is not recoverable. 1 Exception is recoverable. The RI bit indicates whether, from the perspective of the processor, it is safe to continue (that is, processor state data such as that saved to SRR0 is valid). However, it does not guarantee that the interrupted process is recoverable. Exception handlers must look at SRR1[RI] for determination.
31	LE	Little-endian mode enable. 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

1. Full function reserved bits are saved in the SRR1 when an exception occurs; partial function reserved bits are not saved.

The IEEE floating-point exception mode bits (FE0 and FE1) together define whether floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. As shown in *Table 4-4* on page 140, if either FE0 or FE1 are set, Espresso treats exceptions as precise. MSR bits are guaranteed to be written to SRR1 when the first instruction of the exception handler is encountered. For additional details, see the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 4-4. IEEE Floating-Point Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled.
0	1	Imprecise nonrecoverable. For this setting, the Espresso core operates in floating-point precise mode.
1	0	Imprecise recoverable. For this setting, the Espresso core operates in floating-point precise mode.
1	1	Floating-point precise mode.

4.3.4 Enabling and Disabling Exceptions

When a condition exists that might cause an exception to be generated, it must be determined whether the exception is enabled for that condition.

- IEEE-enabled floating-point exceptions (a type of program exception) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either bit is set, all IEEE-enabled floating-point exceptions are taken and cause a program exception.
- Asynchronous, maskable exceptions (such as the external and decremter interrupts) are enabled by setting MSR[EE]. When MSR[EE] = '0', recognition of these exception conditions is delayed. MSR[EE] is cleared automatically when an exception is taken to delay recognition of conditions causing those exceptions.
- A machine check exception can occur only if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs. Individual machine check exceptions can be enabled and disabled through bits in the HID0 Register, which is described in *Table 4-6* on page 144.
- System reset exceptions cannot be masked.

4.3.5 Steps for Exception Processing

After it is determined that the exception can be taken (by confirming that any instruction-caused exceptions occurring earlier in the instruction stream have been handled, and by confirming that the exception is enabled for the exception condition), the processor does the following:

1. SRR0 is loaded with an instruction address that depends on the type of exception. Normally, this is the instruction that would have been completed next had the exception not been taken. See the individual exception description for details about how this register is used for specific exceptions.
2. SRR1[1:4, 10:15] are loaded with information specific to the exception type.
3. SRR1[5:9, 16:31] are loaded with a copy of the corresponding MSR bits. Depending on the implementation, reserved bits might not be copied.
4. The MSR is set as described in *Section 4.3.3* on page 139. The new values take effect as the first instruction of the exception-handler routine is fetched.

Note: MSR[IR] and MSR[DR] are cleared for all exception types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the exception-handler routine.

5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the exception type. The location is determined by adding the exception's vector (see *Table 4-2* on page 134) to the base address determined by MSR[IP]. If IP is cleared, exceptions are vectored to the physical address x'000n_nnnn'. If IP is set, exceptions are vectored to the physical address x'FFFn_nnnn'. For a machine check exception that occurs when MSR[ME] = '0' (machine check exceptions are disabled), the check-stop state is entered (the machine stops executing instructions).

4.3.6 Setting MSR[RI]

The RI bit in the MSR indicates to the exception handler whether the exception is recoverable. When an exception occurs, the RI bit is copied from the MSR to SRR1 and cleared in the MSR. All interrupts are disabled except machine check. If a machine check exception occurs while MSR[RI] is clear, a zero value is found in SRR1[RI] to indicate that the machine state is definitely not recoverable. When this bit is a one, the exception is recoverable as far as the current state of the machine and all programs are concerned including noncritical machine checks. An operating system might handle MSR[RI] as follows:

- In all exceptions, if SRR1[RI] is cleared, the machine state is not recoverable. If it is set, the exception is recoverable with respect to the processor and all programs.
- Use the SPRG0 - SPRG3 Registers to aid in saving the machine state. The following approach is recommended:
 - Have SPRG0 pointing to a stack-save area in memory; save three GPRs in SPRG1 - 3.
 - Move SPRG0 into one of the GPRs that was saved. This GPR now points to the save area in memory.
 - Move the GPRs, SRR0, SRR1, SPRG1 - 3, and other registers to be used by the exception routine into the stack save area.
 - Update SPRG0 to point to a new save area.
 - Set MSR[RI] to indicate that machine state has been saved.
 - Also set MSR[EE] if you want to reen able external interrupts.
- When exception processing is complete:
 - Clear MSR[EE] and MSR[RI].
 - Adjust SPRG0 to point to the stack saved area.
 - Restore the GPRs, SRR0, SRR1, and any other register that you might have saved.
 - Execute **rfi**. This returns the processor to the interrupted program.

4.3.7 Returning from an Exception Handler

The Return from Interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. In general, execution of the **rfi** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an exception. If a previous instruction causes a direct-store interface error exception, the results must be determined before this instruction is executed.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfi** instruction copies SRR1 bits back into the MSR.
- Instructions fetched after this instruction execute in the context established by this instruction.
- Program execution resumes at the instruction indicated by SRR0.

For a complete description of context synchronization, see the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

4.4 Process Switching

The following instructions are useful for restoring the correct context during process switching:

- The **sync** instruction orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes. For an example showing use of **sync**, see the PowerPC register set section of the *PowerPC Microprocessor Family: The Programming Environments* manual.
- The **isync** instruction waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, and protection) established by the previous instructions.
- The **stwcx.** instruction clears any outstanding reservations, ensuring that an **lwarx** instruction in an old process is not paired with an **stwcx.** instruction in a new one.

The operating system should set MSR[RI] as described in *Section 4.3.6 Setting MSR[RI]* on page 142.

4.5 Exception Definitions

Table 4-5 shows all the types of exceptions that can occur in the Espresso microprocessor, and the MSR settings when the processor goes into supervisor mode due to an exception. Depending on the exception, some of these bits are stored in the SRR1 when an exception is taken.

Table 4-5. MSR Setting Due to Exception

Exception Type	MSR Bit ¹															
	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	PM	RI	LE
System reset	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Machine check	0	—	0	0	0	0	0	0	0	0	—	0	0	0	0	ILE
DSI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
ISI	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
External interrupt	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Alignment	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Program	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Floating-point unavailable	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Decrementer interrupt	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
System call	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Trace exception	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
Performance monitor	0	—	0	0	0	—	0	0	0	0	—	0	0	0	0	ILE
¹ 0 Bit is cleared. ILE Bit is copied from the MSR[ILE]. — Bit is not altered. Reserved bits are read as if written as 0.																

The setting of the exception prefix bit (IP) determines how exceptions are vectored. If the bit is cleared, exceptions are vectored to the physical address $x'000n_nnnn'$ (where $nnnn$ is the vector offset); if IP is set, exceptions are vectored to physical address $x'FFFn_nnnn'$. *Table 4-2* on page 134 shows the exception vector offset of the first instruction of the exception handler routine for each exception type.

4.5.1 .Machine Check Exception (x'00200')

The Espresso core implements the machine check exception as defined in the PowerPC Architecture (OEA). It conditionally initiates a machine check exception after one of the following events occurs:

- After DMA look-up missed in the locked cache
- After a **dcbz_I** hit in the normal cache
- After the machine check interrupt ($\overline{MCP_Cx}$) signal is asserted

As defined in the OEA, the exception is not taken if MSR[ME] is cleared, in which case the processor enters a checkstop state.

Certain machine check conditions can be enabled and disabled using HID0 bits, as described in *Table 4-6*.

Table 4-6. HID0 Machine Check Enable Bits

Bit	Name	Function
0	EMCP	Enable $\overline{MCP}$. The primary purpose of this bit is to mask out additional machine check exceptions caused by the assertion of $\overline{MCP_Cx}$, similar to how MSR[EE] can mask external interrupts.
		0 Masks $\overline{MCP}$. Asserting $\overline{MCP_Cx}$ does not generate a machine check exception or a checkstop.
		1 Asserting $\overline{MCP_Cx}$ causes a checkstop if MSR[ME] = '0' or a machine check exception if MSR[ME] = '1'.
15	NHR	Not hard reset (software use only).
		0 A hard reset occurred if software had previously set this bit.
		1 A hard reset has not occurred.

If MSR[ME] and the appropriate HID0 bits are set, the exception is recognized and handled; otherwise, the processor generates an internal checkstop condition. When the exception is recognized, all incomplete stores are discarded. The bus protocol operates normally.

A machine check exception can result from referencing a nonexistent physical address, either directly (with MSR[DR] = '0') or through an invalid translation. If a **dcbz** instruction introduces a block into the cache associated with a nonexistent physical address, a machine check exception can be delayed until an attempt is made to store that block to main memory. In the case of a misaligned load or store that crosses the high memory address boundary ($x'FFFFFFFF$) in real addressing mode, the load/store unit wraps the second part of the operand to address $x'00000000$, and no exception is taken. The specification of such a load or store is considered a programming error.

Machine check exceptions are enabled when MSR[ME] = '1'; this is described in *Section 4.5.1.1*. If MSR[ME] = '0' and a machine check occurs, the processor enters the checkstop state. The checkstop state is described in *Section 4.5.1.2 Checkstop State (MSR[ME] = '0')*.

4.5.1.1 Machine Check Exception Enabled (MSR[ME] = '1')

Machine check exceptions are enabled when MSR[ME] = '1'. When a machine check exception is taken, registers are updated as shown in *Table 4-7*.

Table 4-7. Machine Check Exception (Register Settings)

Register	Setting Description			
SRR0	On a best-effort basis, the Espresso core can set this register to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.			
SRR1	0:9	Cleared.		
	10	Set when a DMA or locked cache error happens.		
	11	Cleared.		
	12	Set when the $\overline{\text{MCP_Cx}}$ signal is asserted; otherwise, zero.		
	13:15	Cleared.		
	16:31	MSR[16:31].		
MSR	POW	0	FP	0
	ILE	—	ME	0
	EE	0	FE0	0
	PR	0	SE	0
			BE	0
			FE1	0
			IP	—
			IR	0
			DR	0
			PM	0
			RI	0
			LE	Set to value of ILE

Note: To handle another machine check exception, the exception handler should set MSR[ME] as soon as it is practical after a machine check exception is taken. Otherwise, subsequent machine check exceptions cause the processor to enter the checkstop state.

The machine check exception is usually unrecoverable in the sense that execution cannot resume in the context that existed before the exception (see *Section 4.3.6 Setting MSR[RI]* on page 142). If the condition that caused the machine check does not otherwise prevent continued execution, MSR[ME] is set to allow the processor to continue execution at the machine check exception vector address and prevent the processor from entering checkstop state if another machine check occurs. Typically, earlier processes cannot resume; however, operating systems can use the machine check exception handler to try to identify and log the cause of the machine check condition.

When a machine check exception is taken, instruction fetching resumes at offset x'00200' from the physical base address indicated by MSR[IP].

4.5.1.2 Checkstop State (MSR[ME] = '0')

If MSR[ME] = '0' and a machine check occurs, the processor enters the checkstop state. The Espresso core can also be forced into the checkstop state by the assertion of the CKSTP_IN primary input signal.

When a processor is in the checkstop state, instruction processing is suspended and generally cannot resume without the processor being reset. The contents of all latches are frozen within two cycles upon entering the checkstop state.

4.5.2 DSI Exception (x'00300')

A DSI exception occurs when no higher priority exception exists and an error condition related to a data memory access occurs. The DSI exception is implemented as it is defined in the PowerPC Architecture (OEA). In case of a TLB miss for a load, store, or cache operation, a DSI exception is taken if the resulting hardware table search causes a page fault.

In Espresso, a DSI exception is taken when a load or store is attempted to a direct-store segment (SR[T] = '1'). In Espresso, a floating-point load or store to a direct-store segment causes a DSI exception rather than an alignment exception, as specified by the PowerPC Architecture.

Espresso also implements the data address breakpoint facility, which is defined as optional in the PowerPC Architecture and is supported by the optional data address breakpoint register (DABR). Although the architecture does not strictly prescribe how this facility must be implemented, the Espresso core follows the recommendations provided by the architecture and described in the *Section 2 Programming Model* on page 49 and the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

4.5.3 ISI Exception (x'00400')

An ISI exception occurs when no higher priority exception exists and an attempt to fetch the next instruction fails. This exception is implemented as it is defined by the PowerPC Architecture (OEA), and is taken for the following conditions:

- The effective address cannot be translated.
- The fetch access is to a no-execute segment (SR[N] = '1').
- The fetch access is to guarded storage and MSR[IR] = '1'.
- The fetch access is to a segment for which SR[T] is set.
- The fetch access violates memory protection.

When an ISI exception is taken, instruction fetching resumes at offset x'00400' from the physical base address indicated by MSR[IP].

4.5.4 External Interrupt Exception (x'00500')

An external interrupt is signaled to the processor by the assertion of the external interrupt signal ($\overline{\text{INT_Cx}}$ where where x = 0 for core 0, 1 for core 1, or 2 for core 2). The $\overline{\text{INT_Cx}}$ signal is expected to remain asserted until the Espresso core takes the external interrupt exception. If $\overline{\text{INT_Cx}}$ is negated early, recognition of the interrupt request is not guaranteed. After Espresso begins execution of the external interrupt handler, the system can safely negate the $\overline{\text{INT_Cx}}$ signal. When Espresso detects the assertion of $\overline{\text{INT_Cx}}$, it stops dispatching and waits for all pending instructions to complete. This allows any instructions in progress that have to take an exception to do so before the external interrupt is taken. After all instructions have vacated the completion buffer, Espresso takes the external interrupt exception as defined in the PowerPC Architecture (OEA).

An external interrupt can be delayed by other higher priority exceptions or if MSR[EE] is cleared when the exception occurs. Register settings for this exception are described in the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

When an external interrupt exception is taken, instruction fetching resumes at offset x'00500' from the physical base address indicated by MSR[IP].

4.5.5 Alignment Exception (x'00600')

The Espresso core implements the alignment exception as defined by the PowerPC Architecture (OEA). An alignment exception is initiated when any of the following events occur:

- The operand of a floating-point load or store is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, or **stwcx**. is not word-aligned.
- The operand of **dcbz** or **dcbz_l** is in a page that is write-through or cache-inhibited.
- An attempt is made to execute **dcbz** or **dcbz_l** when the data cache is disabled.
- An **eciwx** or **ecowx** is not word-aligned.
- A multiple or string access is attempted with MSR[LE] set.

Note: In the Espresso core, the paired-single quantization load or store generates an alignment exception if the operand is not word-aligned when the corresponding GQRn[LD_TYPE] or GQRn[ST_TYPE] are type 0. It does not generate an alignment exception when the corresponding GQRn[LD_TYPE] or GQRn[ST_TYPE] are 4, 5, 6, or 7. Also, a floating-point load or store to a direct-store segment causes a DSI exception rather than an alignment exception, as specified by the PowerPC Architecture. For more information, see *Section 4.5.2 DSI Exception (x'00300')* on page 145.

Execution of paired-single load or store instructions is incompatible with little-endian mode. If enabled, Espresso takes an alignment exception if an attempt is made to execute a paired-single load or store instruction in little-endian mode to avoid the data corruption that otherwise might occur. This alignment exception is enabled by setting HID4[LPE] = '1'.

Also note that Espresso does not raise an alignment exception in the case of a misaligned load or store instruction that crosses the high memory address boundary (x'FFFFFFFF') in real addressing mode. Such an instruction accesses low memory (x'00000000') for the second part of its operand. Use of such an instruction is considered a programming error.

4.5.6 Program Exception (x'00700')

The Espresso core implements the program exception as defined by the PowerPC Architecture (OEA). A program exception occurs when no higher priority exception exists and one or more of the exception conditions defined in the OEA occur.

The Espresso core invokes the system illegal instruction program exception when it detects any instruction from the illegal instruction class. Espresso fully decodes the SPR field of the instruction. If an undefined SPR is specified, a program exception is taken.

The UISA defines **mtspr** and **mfspir** with the record bit (Rc) set as causing a program exception or giving a boundedly-undefined result. In the Espresso core, the appropriate Condition Register (CR) should be treated as undefined. Likewise, the PowerPC Architecture states that the Floating Compared Unordered (**fcmpu**) or Floating Compared Ordered (**fcmpo**) instruction with the record bit set can either cause a program exception or provide a boundedly-undefined result. In Espresso, the BF field in an instruction encoding for these cases is considered undefined.

The Espresso core does not support either of the two floating-point imprecise modes supported by the PowerPC Architecture. Unless exceptions are disabled (MSR[FE0] = '0', MSR[FE1] = '0'), all floating-point exceptions are treated as precise.

When a program exception is taken, instruction fetching resumes at offset x'00700' from the physical base address indicated by MSR[IP]. The exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual describes the register settings for this exception.

4.5.7 Floating-Point Unavailable Exception (x'00800')

The floating-point unavailable exception is implemented as defined by the PowerPC Architecture. A floating-point unavailable exception occurs when no higher priority exception exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is disabled, (MSR[FP] = '0'). Register settings for this exception are described in the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a floating-point unavailable exception is taken, instruction fetching resumes at offset x'00800' from the physical base address indicated by MSR[IP].

4.5.8 Decrementer Exception (x'00900')

The decrementer exception is implemented in the Espresso core as defined by the PowerPC Architecture. The decrementer exception occurs when no higher priority exception exists, a decrementer exception condition occurs (for example, the Decrementer Register has completed decrementing), and MSR[EE] = '1'. In Espresso, the decrementer register is decremented at one fourth the bus clock rate. Register settings for this exception are described in the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a decrementer exception is taken, instruction fetching resumes at offset x'00900' from the physical base address indicated by MSR[IP].

4.5.9 System Call Exception (x'00C00')

A system call exception occurs when a System Call (**sc**) instruction is executed. In the Espresso core, the system call exception is implemented as it is defined in the PowerPC Architecture. Register settings for this exception are described in the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

When a system call exception is taken, instruction fetching resumes at offset x'00C00' from the physical base address indicated by MSR[IP].

4.5.10 Trace Exception (x'00D00')

The trace exception is taken if MSR[BE] = '1' and the currently completing instruction is a branch or if MSR[SE] = '1'. Each instruction considered during trace mode completes before a trace exception is taken.

Implementation Note: The Espresso core diverges from the PowerPC Architecture in that it does not take trace exceptions on the **isync** instruction.

When a trace exception is taken, instruction fetching resumes as offset x'00D00' from the base address indicated by MSR[IP].

4.5.11 Floating-Point Assist Exception (x'00E00')

The optional floating-point assist exception defined by the PowerPC Architecture is not implemented in the Espresso core .

4.5.12 Performance Monitor Interrupt (x'00F00')

The Espresso core provides a performance monitor facility to monitor and count predefined events such as processor clocks, misses in either the instruction cache or the data cache, instructions dispatched to a particular execution unit, mispredicted branches, and other occurrences. The count of such events can be used to trigger the performance monitor exception. The performance monitor facility is not defined by the PowerPC Architecture.

The performance monitor can be used for the following situations:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior must be monitored and studied to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To help system developers bring up and debug their systems.

The performance monitor uses the following SPRs:

- The Performance Monitor Counter Registers (PMC1 - PMC4) are used to record the number of times a certain event has occurred. UPMC1 - UPMC4 provide user-level read access to these registers.
- The Monitor Mode Control Registers (MMCR0 - MMCR1) are used to enable various performance monitor interrupt functions. UMMCR0 - UMMCR1 provide user-level read access to these registers.
- The Sampled Instruction Address Register (SIA) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The USIA Register provides user-level read access to the SIA.

Table 4-8 lists register settings when a performance monitor interrupt exception is taken.

Table 4-8. Performance Monitor Interrupt Exception (Register Settings)

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0	Loaded with the equivalent MSR bits.		
	1:4	Cleared.		
	5:9	Loaded with the equivalent MSR bits.		
	10:15	Cleared.		
	16:31	Loaded with the equivalent MSR bits.		
MSR	POW	0	FP	0
	ILE	—	ME	—
	EE	0	FE0	0
	PR	0	SE	0
			BE	0
			FE1	0
			IP	—
			IR	0
			DR	0
			PM	0
			RI	0
			LE	Set to value of ILE

As with other PowerPC exceptions, the performance monitor interrupt follows the normal PowerPC exception model with a defined exception vector offset (x'00F00'). The priority of the performance monitor interrupt lies between the external interrupt and the decremter interrupt (see Table 4-3 on page 137). The contents of the SIA are described in Section 2.1.2.3 Performance Monitor Registers on page 57. The performance monitor is described in Section 9 Performance Monitor on page 243.

4.5.13 Instruction Address Breakpoint Exception (x'01300')

An instruction address breakpoint interrupt occurs when the following conditions are met:

- The instruction breakpoint address IABR[0:29] matches EA[0:29] of the next instruction to complete in program order. The instruction that triggers the instruction address breakpoint exception is not executed before the exception handler is invoked.
- The translation enable bit (IABR[TE]) matches MSR[IR].
- The breakpoint enable bit (IABR[BE]) is set. The address match is also reported to the JTAG/COP block, which might subsequently generate a soft or hard reset. The instruction tagged with the match does not complete before the breakpoint exception is taken.

Table 4-9 lists register settings when an instruction address breakpoint exception is taken.

Table 4-9. Instruction Address Breakpoint Exception (Register Settings)

Register	Setting Description			
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no exception conditions were present.			
SRR1	0	Loaded with the equivalent MSR bits.		
	1:4	Cleared.		
	5:9	Loaded with the equivalent MSR bits.		
	10:15	Cleared.		
	16:31	Loaded with the equivalent MSR bits.		
MSR	POW	0	FP	0
	ILE	—	ME	—
	EE	0	FE0	0
	PR	0	SE	0
			BE	0
			FE1	0
			IP	—
			IR	0
			DR	0
			PM	0
			RI	0
			LE	Set to value of ILE

The Espresso core requires that an **mtspr** to the IABR be followed by a context-synchronizing instruction. Espresso cannot generate a breakpoint response for that context-synchronizing instruction if the breakpoint is enabled by the **mtspr(IABR)** immediately preceding it. Espresso also cannot block a breakpoint response on the context-synchronizing instruction if the breakpoint was disabled by the **mtspr(IABR)** instruction immediately preceding it. The format of the IABR is shown in *Section 2.1.2.1 Instruction Address Breakpoint Register (IABR)* on page 55.

When an instruction address breakpoint exception is taken, instruction fetching resumes as offset x'01300' from the base address indicated by MSR[IP].

5. Memory Management

This section describes the Espresso microprocessor implementation of the memory management unit (MMU) specifications provided by the operating environment architecture (OEA) for PowerPC processors. The primary function of the MMU in a PowerPC processor is the translation of logical (effective) addresses to physical addresses (referred to as real addresses in the architecture specification) for memory accesses and I/O accesses (I/O accesses are assumed to be memory-mapped). In addition, the MMU provides access protection on a segment, block, or page basis. This section describes the specific hardware used to implement the MMU model of the OEA in the Espresso microprocessor. See the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete description of the conceptual model.

Note: The Espresso microprocessor does not implement the optional direct-store facility, and it is not likely to be supported in future designs.

Two general types of memory accesses generated by PowerPC processors require address translation: instruction accesses and data accesses generated by load and store instructions. Generally, the address translation mechanism is defined in terms of the segment descriptors and page tables that PowerPC processors use to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the interim virtual address to a physical address.

The segment descriptors, used to generate the interim virtual addresses, are stored as on-chip segment registers on 32-bit implementations (such as Espresso). In addition, two translation lookaside buffers (TLBs) are implemented on Espresso to keep recently-used page address translations on-chip. Although the PowerPC OEA describes one MMU (conceptually), the Espresso hardware maintains separate TLBs and table search resources for instruction and data accesses that can be performed independently (and simultaneously). Therefore, Espresso is described as having two MMUs, one for instruction accesses (IMMU) and one for data accesses (DMMU).

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs). There are separate instruction and data BAT mechanisms that reside in the instruction and data MMUs.

The MMUs, together with the exception processing mechanism, provide the necessary support for the operating system to implement a paged virtual memory environment and to enforce protection of designated memory areas.

Exception processing is described in *Section 4 Exceptions* on page 133. Specifically, *Section 4.3 Exception Processing* on page 138 describes the Machine State Register (MSR), which controls some of the critical functionality of the MMUs.

5.1 MMU Overview

The Espresso microprocessor implements the memory management specification of the PowerPC OEA for 32-bit implementations. Thus, it provides 4 GB of effective address space accessible to supervisor and user programs, with a 4 KB page size (or 128 KB page size in large page mode) and a 256 MB segment size. In addition, the MMUs of 32-bit PowerPC processors use an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit physical addresses. PowerPC processors also have a BAT mechanism for mapping large blocks of memory. Block sizes range from 128 KB to 256 MB and are software-programmable.

Basic features of the Espresso MMU implementation defined by the OEA are as follows:

- Support for real addressing mode. Effective-to-physical address translation can be disabled separately for data and instruction accesses.
- Block address translation. Each of the BAT array entries (eight IBAT entries and eight DBAT entries) provides a mechanism for translating blocks as large as 256 MB from the 32-bit effective address space into the physical memory space. This can be used for translating large address ranges whose mappings do not change frequently.
- Segmented address translation. The 32-bit effective address is extended to a 52-bit virtual address by substituting 24 bits of upper address bits from the segment register for the 4 upper bits of the effective address (EA), which are used as an index into the segment register file. This 52-bit virtual address space is divided into 4 KB or 128 KB pages, each of which can be mapped to a physical page.

The Espresso MMU also provides the following features that are not required by the PowerPC Architecture:

- 4 KB or 128 KB page size selectable at startup.
- Separate translation lookaside buffers (TLBs). The 128-entry, 2-way set-associative ITLBs and DTLBs keep recently used page address translations on-chip.
- Table search operations performed in hardware. The 52-bit virtual address is formed and the MMU attempts to fetch the PTE, which contains the physical address, from the appropriate TLB on-chip. If the translation is not found in a TLB (that is, a TLB miss occurs), the hardware performs a table search operation (using a hashing function) to search for the PTE.
- TLB invalidation. Espresso implements the optional TLB Invalidate Entry (**tlbie**) and TLB Synchronize (**tlbsync**) instructions, which can be used to invalidate TLB entries. For more information about the **tlbie** and **tlbsync** instructions, see *Section 5.4.3.2 TLB Invalidation* on page 176.

Table 5-1 summarizes the Espresso MMU features, including those defined by the PowerPC Architecture (OEA) for 32-bit processors and those specific to the Espresso implementation.

Table 5-1. MMU Feature Summary (Page 1 of 2)

Feature Category	Architecturally Defined/ Espresso-Specific	Feature
Address ranges	Architecturally defined	2 ³² bytes of effective address.
		2 ⁵² bytes of virtual address.
		2 ³² bytes of physical address.
Page size	Architecturally defined	4 KB.
	Espresso-specific	128 KB.
Segment size	Architecturally defined	256 MB.

Table 5-1. MMU Feature Summary (Page 2 of 2)

Feature Category	Architecturally Defined/ Espresso-Specific	Feature
Block address translation	Architecturally defined	Range of 128 KB - 256 MB sizes.
		Implemented with IBAT and DBAT registers in BAT array.
Memory protection	Architecturally defined	Segments selectable as no-execute.
		Pages selectable as user/supervisor and read-only or guarded.
		Blocks selectable as user/supervisor and read-only or guarded.
Page history	Architecturally defined	Referenced and changed bits defined and maintained.
Page address translation	Architecturally defined	Translations stored as PTEs in hashed page tables in memory.
		Page table size determined by mask in SDR1 Register.
TLBs	Architecturally defined	Instructions for maintaining TLBs (tlbie and tlbsync instructions in Espresso).
	Espresso-specific	128-entry, 2-way set associative ITLB. 128-entry, 2-way set associative DTLB. LRU replacement algorithm.
Segment descriptors	Architecturally defined	Stored as segment registers on-chip (two identical copies maintained).
Page table search support	Espresso-specific	Espresso performs the table search operation in hardware.

5.1.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual, augmented with information in this section. The memory subsystem uses the physical address for the access.

For a complete discussion of effective address calculation, see *Section 2.3.2.3 Effective Address Calculation* on page 75.

5.1.2 MMU Organization

Figure 5-1 on page 155 shows the conceptual organization of a PowerPC MMU in a 32-bit implementation; note that it does not describe the specific hardware used to implement the memory management function for a particular processor. Processors can optionally implement on-chip TLBs, hardware support for the automatic search of the page tables for PTEs, and other hardware features (invisible to the system software) not shown. This, and each of the following figures, assume the use of 4 KB pages. In large page mode, five fewer bits of the effective address are translated (EA[0:14]). Five additional bits of the effective address become part of the page offset (EA[15:31]) and, therefore, become the low-order bits of the physical address (see *Section 5.1.4 Large Page Support* on page 160).

Espresso maintains two on-chip TLBs with the following characteristics:

- 128 entries, 2-way set associative (64×2), least recently used (LRU) replacement.
- Data TLB supports the DMMU; instruction TLB supports the IMMU.
- Hardware TLB update.
- Hardware update of referenced (R) and changed (C) bits in the translation table.

In the event of a TLB miss, the hardware attempts to load the TLB based on the results of a translation table search operation.

Figure 5-2 on page 156 shows the conceptual organization of the Espresso instruction MMU. The instruction addresses shown in *Figure 5-2* are generated by the processor for sequential instruction fetches and addresses that correspond to a change of program flow. *Figure 5-3* on page 157 shows the conceptual organization of the Espresso data MMU. Data addresses shown in *Figure 5-3* are generated by load, store, and cache instructions.

As shown in the figures, after an address is generated, the high-order bits of the effective address, EA[0:19] (or a smaller set of address bits, EA[0:*n*], in the case of blocks or large pages), are translated into physical address bits PA[0:19]. The low-order address bits, A[20:31], are untranslated and are therefore identical for both effective and physical addresses. After translating the address, the MMUs pass the resulting 32-bit physical address to the memory subsystem. The MMUs record whether the translation is for an instruction or data access, whether the processor is in user or supervisor mode, and, for data accesses, whether the access is a load or a store operation.

The MMUs use this information to appropriately direct the address translation and to enforce the protection hierarchy programmed by the operating system. *Section 4.3 Exception Processing* on page 138 describes the MSR, which controls some of the critical functionality of the MMUs.

The figures show how address bits A[20:26] index into the on-chip instruction and data caches to select a cache set. The remaining physical address bits are then compared with the tag fields (comprised of bits PA[0:19]) of the two selected cache blocks to determine if a cache hit has occurred. In the case of a cache miss, the instruction or data access is then forwarded to the L2 tags to check for an L2 cache hit. In case of a miss, the access is forwarded to the bus interface unit, which initiates an external memory access.

Figure 5-1. MMU Conceptual Block Diagram

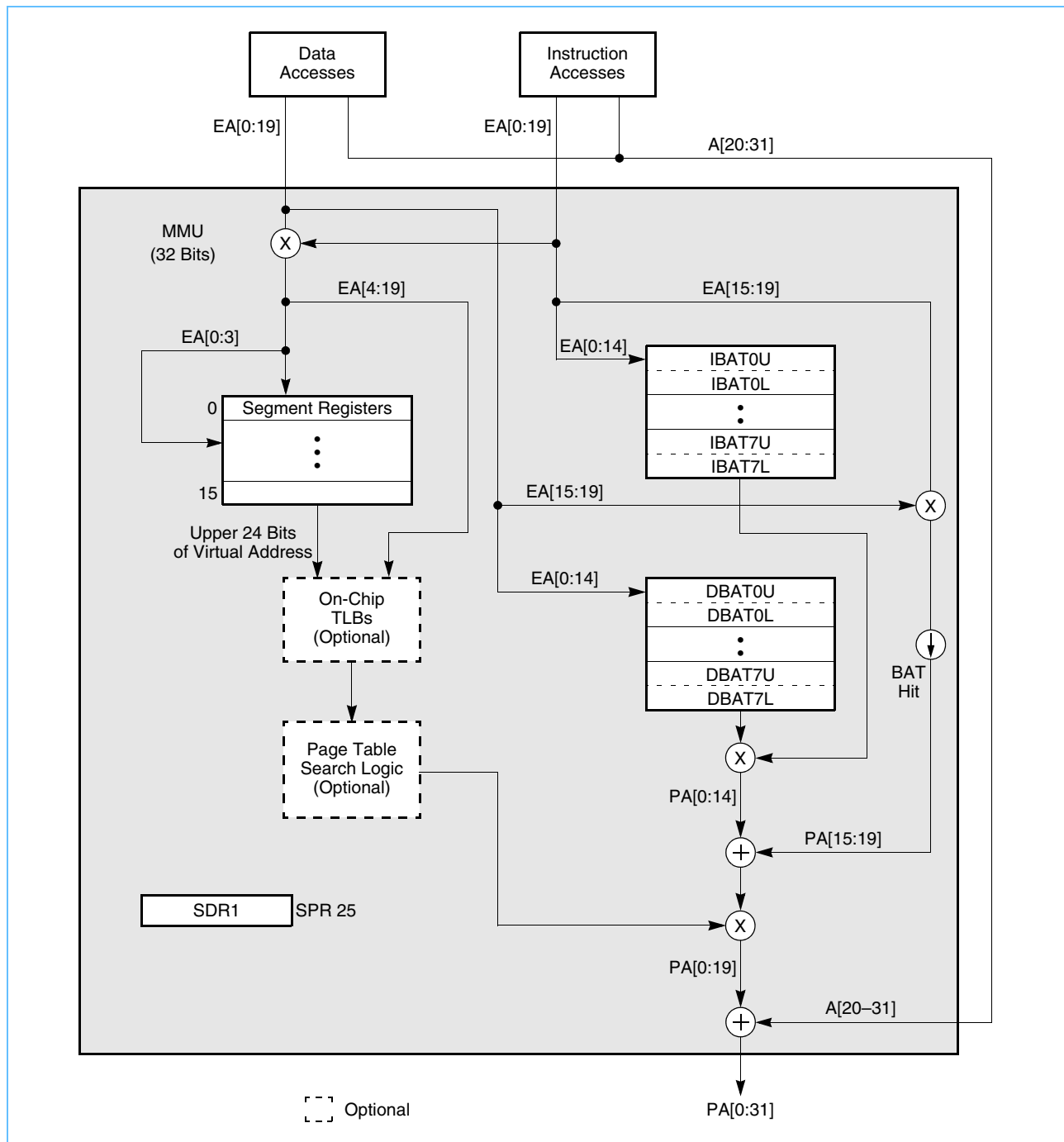


Figure 5-2. Espresso Microprocessor IMMU Block Diagram

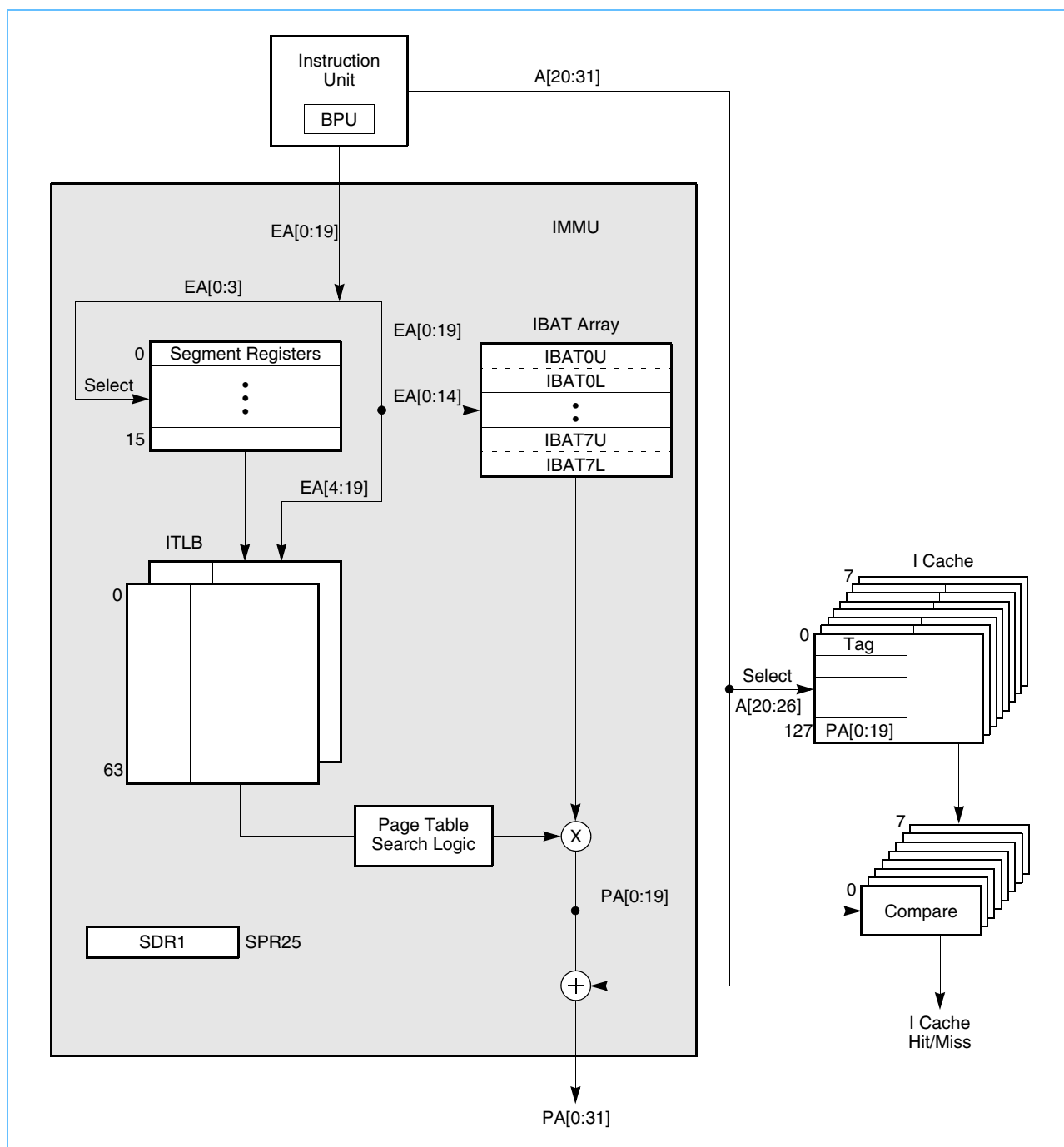
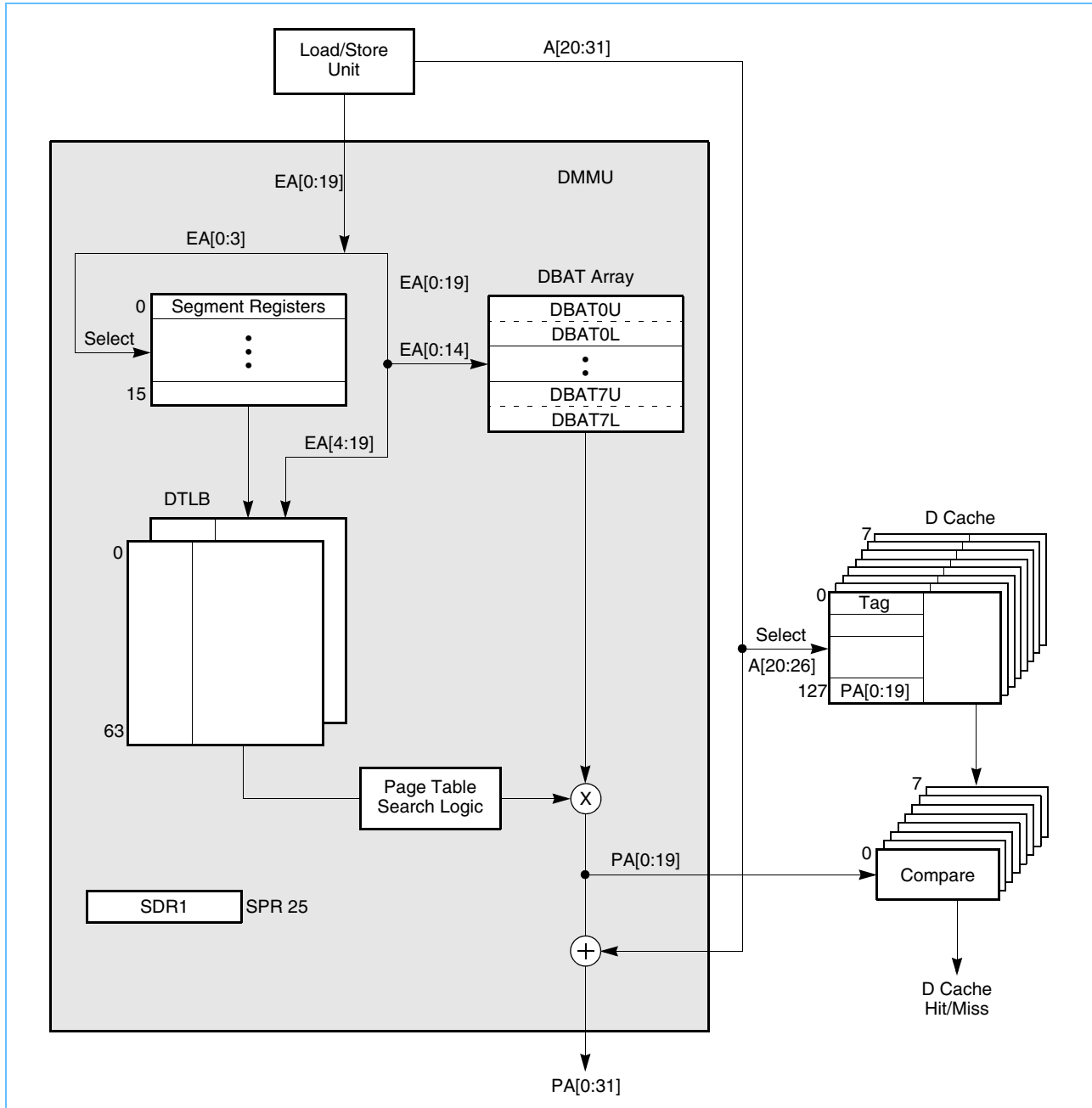


Figure 5-3. Espresso Microprocessor DMMU Block Diagram



5.1.3 Address Translation Mechanisms

PowerPC processors support the following three types of address translation:

- Page address translation. Translates the page frame address for a 4 KB or 128 KB page size.
- Block address translation. Translates the block number for blocks that range in size from 128 KB - 256 MB.
- Real addressing mode address translation. When address translation is disabled, the physical address is identical to the effective address.

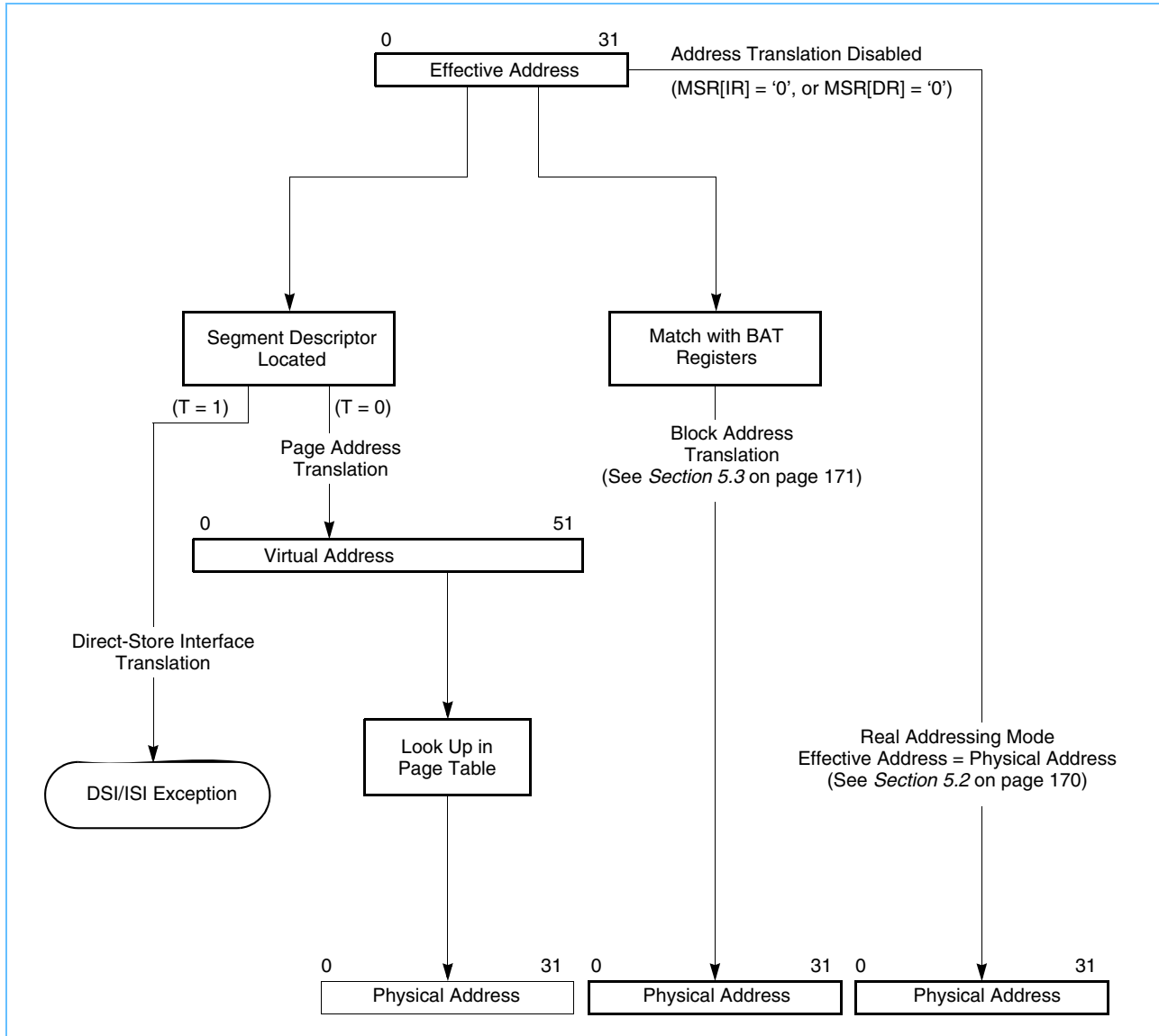
Figure 5-4 on page 159 shows the three address translation mechanisms provided by the MMUs. The segment descriptors shown in the figure control the page address translation mechanism. When an access uses page address translation, the appropriate segment descriptor is required. In 32-bit implementations, the appropriate segment descriptor is selected from the 16 on-chip segment registers by the 4 highest-order effective address bits.

A control bit in the corresponding segment descriptor then determines if the access is to memory (memory-mapped) or to the direct-store interface space. Note that the direct-store interface was present in the architecture only for compatibility with existing I/O devices that used this interface. However, it is being removed from the architecture, and the Espresso microprocessor does not support it. When an access is determined to be to the direct-store interface space, the Espresso microprocessor takes a DSI exception if it is a data access (see Section 4.5.2 *DSI Exception (x'00300')* on page 145), and takes an ISI exception if it is an instruction access (see Section 4.5.3 *ISI Exception (x'00400')* on page 146).

For memory accesses translated by a segment descriptor, the interim virtual address is generated using the information in the segment descriptor. Page address translation corresponds to the conversion of this virtual address into the 32-bit physical address used by the memory subsystem. In most cases, the physical address for the page resides in an on-chip TLB and is available for quick access. However, if the page address translation misses in the on-chip TLB, the MMU causes a search of the page tables in memory (using the virtual address information and a hashing function) to locate the required physical address.

Because blocks are larger than pages, there are fewer upper-order effective address bits to be translated into physical address bits for block address translation. More low-order address bits (at least 17) are untranslated to form the offset into a block. Also, instead of segment descriptors and a TLB, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding field of a BAT register, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored.

Figure 5-4. Address Translation Types



When the processor generates an access, and the corresponding address translation enable bit in the MSR is cleared, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. Instruction address translation is enabled by setting MSR[IR], and data address translation is enabled by setting MSR[DR].

5.1.4 Large Page Support

The previous design supported only a 4 KB page size for address translation and protection. The Espresso core supports either a 4 KB page size or a 128 KB page size, selectable at startup. Larger pages have the advantage of fewer page table entries needed to translate for a given address space. This in turn allows a reduction in the page table size, and fewer TLB misses.

The 32-bit effective address is translated into a 32-bit real address (RA) in two steps:

1. The EA[0:3] is indexed into one of 16 segment registers to obtain a virtual segment ID. When the virtual segment ID is concatenated with the rest of the EA, it forms the 52-bit virtual address (VA).
2. The virtual page number (VPN) is then translated into a real page number (RPN) through the TLB. The low-order bits of the address are the page offset, and are the same for the EA and RA.

For a 4 KB page size, the low-order 12 bits comprise the page offset, and EA[20:31] is concatenated to the RPN to get the RA. To support a larger page size, additional low-order bits become page offset bits, and the page-index field shrinks by a corresponding amount. *Table 5-2* identifies the sizes and EA bit ranges of various address fields for the small-page and large-page sizes. The TLB index is composed of the six lowest-order bits of the page index.

Table 5-2. Page Index and Offset Bits for Small and Large Pages

Page Size	SR Index	Page Index	Page Offset	TLB Index
4 KB	EA[0:3]	EA[4:19]	EA[20:31]	EA[14:19]
128 KB	EA[0:3]	EA[4:14]	EA[15:31]	EA[9:14]

The content of a given TLB entry includes the virtual page number (VPN) consisting of the 24-bit virtual segment ID (VSID) and 16-bit page index (for 4 KB pages), the corresponding RPN, a valid bit, and protection and memory attributes. A TLB hit occurs when one of the two entries selected by the TLB index has its valid bit asserted, and the corresponding VPN in that entry matches the VPN associated with the current access.

For larger page sizes, the page index comprises fewer bits, and so the VPN is correspondingly smaller. The compare operation therefore is applied to fewer bits in the TLB entry. For the large-page size, the page index is 11 bits, and therefore only the first 35 bits in the VPN are compared.

5.1.4.1 Page Table Lookup Procedure

When a TLB miss occurs, hardware “walks” the page table to search for a page table entry (PTE) that can translate the address. The PowerPC Architecture specifies the use of a hashing function to compute the address in the page table of the PTE group (PTEG) to which the PTE belongs. Since the operating system initially populates the page table with these PTEs, it must use the same hashing function to build the page table. For the 4 KB page size, the hash function is the exclusive-OR of VSID[5:23] with the page index prepended with ‘000’:

$$\text{hash_value} = \text{VSID}[5:23] \text{ XOR } (000 \mid \text{EA}[4:19])$$

The resulting 19-bit hash value is then masked to reflect the page table size, and depending on that masking, comprises some of bits 7 - 25 of the page table entry group (PTEG) address. The high-order bits of the PTEG address are supplied by the SDR1 Register, while the low-order 6 bits are zeros. If the PTE is not found in the primary PTEG, a secondary hash is needed, which is simply the ones complement of the primary hash value.

For larger page sizes, this hash function is modified by eliminating the bits of the page index that have become part of the page offset, and shifting high-order page offset bits in place of the eliminated bits. For 128 KB pages, EA[15:31] is the page offset and EA[4:14] is the page index. To hash the page-index bits with the low-order bits of the VSID, the 19-bit hash value is computed as follows:

$$\text{hash_value} = \text{VSID}[5-23] \text{ XOR } (00000000 \mid \text{EA}[4-14])$$

Because the larger page sizes require correspondingly fewer entries in the page table for a given physical memory size, this definition for the hash value maintains the same number of bits of hashed and unhashed address bits for the page table entry as the 4 KB page size case.

5.1.5 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMUs provide access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute or guarded. *Table 5-3* shows the protection options supported by the MMUs for pages.

Table 5-3. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	Yes	Yes	Yes
Supervisor-only-no-execute	—	—	—	—	Yes	Yes
Supervisor-write-only	Yes	Yes	—	Yes	Yes	Yes
Supervisor-write-only-no-execute	—	Yes	—	—	Yes	Yes
Both (user/supervisor)	Yes	Yes	Yes	Yes	Yes	Yes
Both (user/supervisor) no-execute	—	Yes	Yes	—	Yes	Yes
Both (user/supervisor) read-only	Yes	Yes	—	Yes	Yes	—
Both (user/supervisor) read-only-no-execute	—	Yes	—	—	Yes	—
Yes Access permitted — Protection violation						

The no-execute option provided in the segment register lets the operating system program determine whether instructions can be fetched from an area of memory. The remaining options are enforced based on a combination of information in the segment descriptor and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (MSR[PR] = '0') to access the page. User accesses that map into a supervisor-only page cause an exception.

Finally, a facility in the VEA and OEA allows pages or blocks to be designated as guarded, preventing out-of-order accesses that can cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so accesses do not occur unless they are explicitly required by the program.

For more information see the memory protection facilities section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.1.6 Page History Information

The MMUs of PowerPC processors also define referenced (R) and changed (C) bits in the page address translation mechanism that can be used as history information relevant to the page. The operating system can use these bits to determine which areas of memory to write back to disk when new pages must be allocated in main memory. While these bits are initially programmed by the operating system into the page table, the architecture specifies that they can be maintained either by the processor hardware (automatically) or by some software-assist mechanism.

Implementation Note: When loading the TLB, Espresso checks the state of the changed and referenced bits for the matched PTE. If the referenced bit is not set and the table search operation was initially caused by a load operation or by an instruction fetch, Espresso automatically sets the referenced bit in the translation table. Similarly, if the table search operation is caused by a store operation and either the referenced bit or the changed bit is not set, the hardware automatically sets both bits in the translation table. In addition, when the address translation of a store operation hits in the DTLB, Espresso checks the state of the changed bit. If the bit is not already set, the hardware automatically updates the DTLB and the translation table in memory to set the changed bit. For more information, see *Section 5.4.1 Page History Recording* on page 172.

5.1.7 General Flow of MMU Address Translation

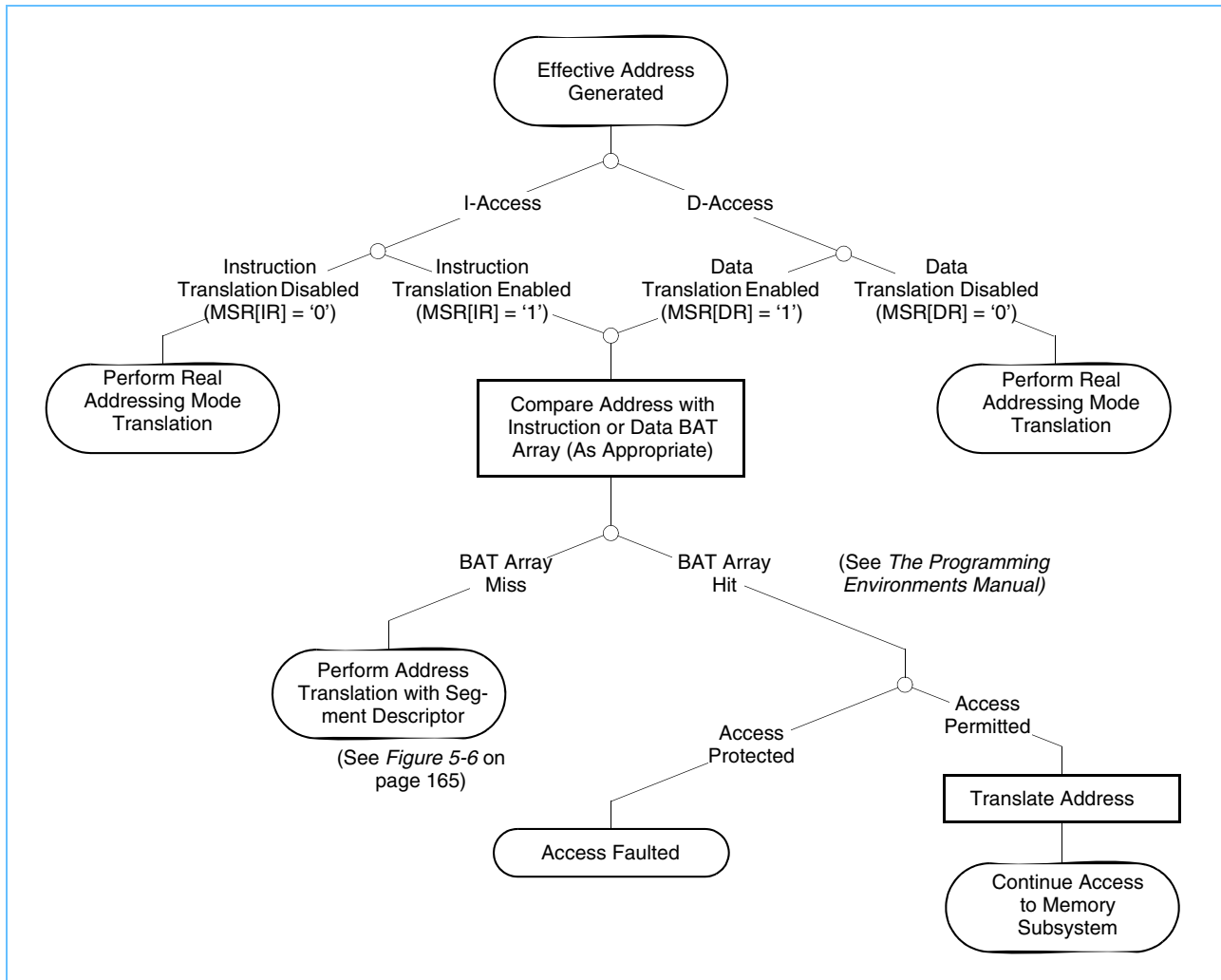
The following sections describe the general flow used by PowerPC processors to translate effective addresses to virtual and then physical addresses.

5.1.7.1 Real Addressing Mode and Block Address Translation Selection

When an instruction or data access is generated and the corresponding instruction or data translation is disabled ($\text{MSR}[\text{IR}] = '0'$ or $\text{MSR}[\text{DR}] = '0'$), real addressing mode is used (physical address equals effective address). The access continues to the memory subsystem as described in *Section 5.2 Real Addressing Mode* on page 170.

Figure 5-5 on page 163 shows the flow the MMUs use in determining whether to select real addressing mode, block address translation, or the segment descriptor to select page address translation.

Figure 5-5. General Flow of Address Translation (Real Addressing Mode and Block)



Note: If the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access violates the protection mechanism, an exception (either ISI or DSI) is generated.

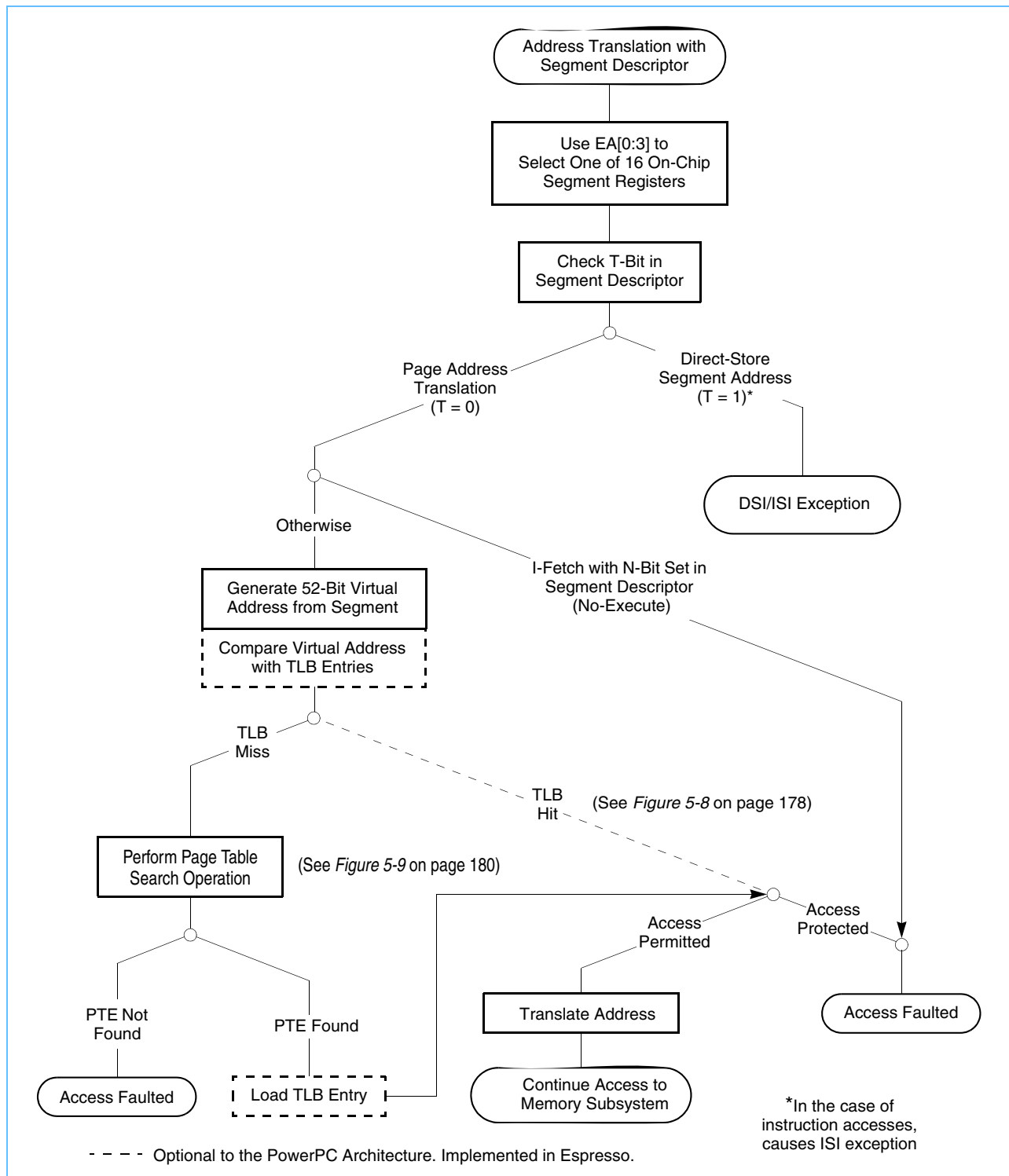
5.1.7.2 Page Address Translation Selection

If address translation is enabled and the effective address information does not match a BAT array entry, the segment descriptor must be located. When the segment descriptor is located, the T bit in the segment descriptor selects whether the translation is to a page or to a direct-store segment as shown in *Figure 5-6 General Flow of Page and Direct-Store Interface Address Translation* on page 165.

For 32-bit implementations, the segment descriptor for an access is contained in one of 16 on-chip segment registers; effective address bits EA[0:3] select one of the 16 segment registers.

Note that Espresso does not implement the direct-store interface, and accesses to these segments cause a DSI or ISI exception. In addition, *Figure 5-6* on page 165 shows how the no-execute protection is enforced; if the N bit in the segment descriptor is set and the access is an instruction fetch, the access is faulted as described in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual. Note that the figure shows the flow for these cases as described by the PowerPC OEA, and so the TLB references are shown as optional. Because Espresso implements TLBs, these branches are valid and are described in more detail throughout this section.

Figure 5-6. General Flow of Page and Direct-Store Interface Address Translation



If $SR[T] = '0'$, page address translation is selected. The information in the segment descriptor is then used to generate the 52-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries [PTEs] in a page table in memory). For increased performance, Espresso has two on-chip TLBs to cache recently-used translations.

If an access hits in the appropriate TLB, page translation succeeds and the physical address bits are forwarded to the memory subsystem. If the required translation is not resident, the MMU performs a search of the page table. If the required PTE is found, a TLB entry is allocated and the page translation is attempted again. This time, the TLB is guaranteed to hit. When the translation is located, the access is qualified with the appropriate protection bits. If the access causes a protection violation, either an ISI or DSI exception is generated.

If the PTE is not found by the table search operation, a page fault condition exists, and an ISI or DSI exception occurs so that software can handle the page fault.

5.1.8 MMU Exceptions Summary

To complete any memory access, the effective address must be translated to a physical address. As specified by the architecture, an MMU exception condition occurs if this translation fails for one of the following reasons:

- Page fault: there is no valid entry in the page table for the page specified by the effective address (and segment descriptor), and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exception conditions defined by the OEA for 32-bit implementations cause either the ISI or the DSI exception to be taken as shown in *Table 5-4*.

Table 5-4. Translation Exception Conditions (Page 1 of 2)

Condition	Description	Exception
Page fault (no PTE found)	No matching PTE found in the page tables and no matching BAT array entry.	I access: ISI exception SRR1[1] = '1'
		D access: DSI exception DSISR[1] = '1'
Block protection violation	See the conditions described for a block in the block memory protection section of the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.	I access: ISI exception SRR1[4] = '1'
		D access: DSI exception DSISR[4] = '1'
Page protection violation	See the conditions described for a page in the page memory protection section of the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.	I access: ISI exception SRR1[4] = '1'
		D access: DSI exception DSISR[4] = '1'
No-execute protection violation	Attempt to fetch an instruction when $SR[N] = '1'$.	ISI exception SRR1[3] = '1'

Table 5-4. Translation Exception Conditions (Page 2 of 2)

Condition	Description	Exception
Instruction fetch from direct-store segment	Attempt to fetch an instruction when SR[T] = '1'.	ISI exception SRR1[3] = '1'
Data access to direct-store segment (including floating-point accesses)	Attempt to perform a load or store (including a floating-point load or store) when SR[T] = '1'.	DSI exception DSISR[5] = '1'
Instruction fetch from guarded memory	Attempt to fetch an instruction when MSR[IR] = '1' and either matching xBAT[G] = '1', or no matching BAT entry and PTE[G] = '1'.	ISI exception SRR1[3] = '1'

The state saved by the processor for each of these exceptions contains information that identifies the address of the failing instruction. See *Section 4 Exceptions* on page 133 for a more detailed description of exception processing.

In addition to the translation exceptions, there are other MMU-related conditions (some of them defined as implementation-specific, and therefore not required by the architecture) that can cause an exception to occur. These exception conditions map to processor exceptions as shown in *Table 5-5*. The only MMU exception conditions that occur when MSR[DR] = '0' are those that cause an alignment exception for data accesses. For more detailed information about the conditions that cause an alignment exception (in particular for string/multiple instructions), see *Section 4.5.5 Alignment Exception (x'00600')* on page 147.

Note: Some exception conditions depend upon whether the memory area is set up as write-through (W = '1') or cache-inhibited (I = '1'). These bits are described fully in the memory/cache access attributes section of the *PowerPC Microprocessor Family: The Programming Environments* manual. See *Section 4 Exceptions* on page 133 and the exceptions section of the *PowerPC Microprocessor Family: The Programming Environments* manual for a complete description of the SRR1 and DSISR bit settings for these exceptions.

Table 5-5. Other MMU Exception Conditions for the Espresso Processor

Condition	Description	Exception
dcbz or dcbz_l with W = '1' or I = '1'	dcbz or dcbz_l instruction to a write-through or cache-inhibited segment or block.	Alignment exception (not required by the architecture for this condition).
lwarx or stwcx. with W = '1'	Reservation instruction to a write-through segment or block.	DSI exception DSISR[5] = '1'
lwarx , stwcx. , eciwX , or ecowX instruction to direct-store segment.	Reservation instruction or external control instruction when SR[T] = '1'.	DSI exception DSISR[5] = '1'
Floating-point load or store to direct-store segment.	Floating-point memory access when SR[T] = '1'.	See data access to direct-store segment in <i>Table 5-4</i> on page 166.
Load or store that results in a direct-store error.	Does not occur in Espresso.	Does not apply.
eciwX or ecowX attempted when external control facility disabled.	eciwX or ecowX attempted with EAR[E] = '0'.	DSI exception DSISR[11] = '1'
lmw , stmw , lswi , lswx , stswi , or stswx instruction attempted in little-endian mode.	lmw , stmw , lswi , lswx , stswi , or stswx instruction attempted when MSR[LE] = '1'.	Alignment exception.
Operand misalignment.	Translation enabled and a floating-point load/store, stmw , stwcx. , lmw , lwarx , eciwX , or ecowX instruction operand is not word-aligned.	Alignment exception (some of these cases are implementation-specific).

5.1.9 MMU Instructions and Register Summary

The MMU instructions and registers allow the operating system to set up the block address translation areas and the page tables in memory.

Note: Because the implementation of TLBs is optional, the instructions that refer to these structures are also optional. However, as these structures serve as caches of the page table, the architecture specifies a software protocol for maintaining coherency between these caches and the tables in memory whenever the tables in memory are modified. When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required. Also note that Espresso implements all TLB-related instructions except **tlbia**, which is treated as an illegal instruction.

Because the MMU specification for PowerPC processors is so flexible, it is recommended that the software that uses these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 5-6 summarizes the Espresso instructions that specifically control the MMU. For more detailed information about the instructions, see *Section 2 Programming Model* on page 49 and the instruction set section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 5-6. Espresso Microprocessor Instruction Summary (Control MMUs) (Page 1 of 2)

Instruction	Description
mtsr SR,rS	Move to Segment Register. $SR[SR\#] \leftarrow rS$ As described in the the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual, follow an mtsr instruction with a context synchronizing instruction (for example, isync) to ensure that succeeding instructions that access the segment registers execute with the updated value.
mtsrin rS,rB	Move to Segment Register Indirect. $SR[rB[0-3]] \leftarrow rS$ As described in the the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual, follow an mtsrin instruction with a context synchronizing instruction (for example, isync) to ensure that succeeding instructions that access the segment registers execute with the updated value.
mfsr rD,SR	Move from Segment Register. $rD \leftarrow SR[SR\#]$
1. These instructions are defined by the PowerPC Architecture, but are optional.	

Table 5-6. Espresso Microprocessor Instruction Summary (Control MMUs) (Page 2 of 2)

Instruction	Description
mfsrin rD,rB	Move from Segment Register Indirect. $rD \leftarrow SR[rB[0-3]]$
tlbie rB ¹	TLB Invalidate Entry. For the effective address specified by rB, $TLB[V] \leftarrow 0$ The tlbie instruction invalidates all TLB entries indexed by the EA. It operates on both the instruction and data TLBs simultaneously invalidating four TLB entries. Software must ensure that instruction fetches or memory references to the virtual pages specified by the tlbie instruction have been completed before executing the tlbie instruction. Note: The architecture specifies that a tlbie instruction must be issued on only one processor at a time in a multi-processor system. The architecture also specifies that a barrier instruction (eieio or sync) is required between a tlbie instruction and a tlbsync instruction. In this implementation, a sync followed by an isync is required between a tlbie instruction and a tlbsync instruction to ensure that the tlbsync instruction is executed after the other cores have had a chance to snoop the TLBIE transaction. Finally, the architecture specifies that a sync instruction should follow the tlbsync instruction to ensure that the tlbsync completes before another core executes a tlbie or tlbsync instruction.
tlbsync ¹	TLB Synchronize. Synchronizes the execution of all other tlbie instructions in the system. In Espresso, the tlbsync instruction completes in one core when there are no pending snooped tlbie instructions in the other two cores.
1. These instructions are defined by the PowerPC Architecture, but are optional.	

Table 5-7 summarizes the registers that the operating system uses to program the Espresso MMUs. These registers are accessible to supervisor-level software only.

These registers are described in *Section 2 Programming Model* on page 49.

Table 5-7. Espresso Microprocessor MMU Registers

Register	Description
Segment registers (SR0 - SR15)	The sixteen 32-bit segment registers are present only in 32-bit implementations of the PowerPC Architecture. The fields in the segment register are interpreted differently depending on the value of bit 0. The segment registers are accessed by the mtsr , mtsrin , mfsr , and mfsrin instructions.
BAT registers (IBAT0U - IBAT7U, IBAT0L - IBAT7L, DBAT0U - DBAT7U and DBAT0L - DBAT7L)	There are 32 BAT registers, organized as eight pairs for instructions and eight pairs for data. The BAT registers are defined as 32-bit registers in 32-bit implementations. These are special-purpose registers that are accessed by the mtspr and mfspr instructions.
SDR1	The SDR1 register specifies the variables used in accessing the page tables in memory. SDR1 is defined as a 32-bit register for 32-bit implementations. This special-purpose register is accessed by the mtspr and mfspr instructions.

5.2 Real Addressing Mode

If address translation is disabled ($\text{MSR}[\text{IR}] = '0'$ or $\text{MSR}[\text{DR}] = '0'$) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as described in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note that the default WIMG bits ('0011') cause data accesses to be considered cacheable ($I = '0'$) and thus load and store accesses are weakly ordered. This is the case even if the data cache is disabled in the HID0 register (as it is after a hard reset). If I/O devices require load and store accesses to occur in strict program order (strongly ordered), translation must be enabled so that the corresponding I bit can be set. Note also, that the G bit must be set to ensure that the accesses are strongly ordered. For instruction accesses, the default memory access mode bits (WIMG) are '0001'. That is, instruction accesses are considered cacheable ($I = '0'$), and the memory is guarded. Again, instruction accesses are considered cacheable even if the instruction cache is disabled in the HID0 register (as it is after a hard reset). The W and M bits have no effect on the instruction cache.

It is considered a programming error to specify a misaligned load or store instruction in real addressing mode that crosses the high memory address boundary (x'FFFFFFFF'). The load/store unit breaks a misaligned access into two separate accesses. In this case, the second access addresses low memory (x'00000000') without raising an exception.

For information about the synchronization requirements for changes to $\text{MSR}[\text{IR}]$ and $\text{MSR}[\text{DR}]$, see *Section 2.3.2.4 Synchronization* on page 75 in this manual and the synchronization requirements for special registers and for lookaside buffers section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.3 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numerical data.

Block address translation is described in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual for 32-bit implementations.

Espresso has an enhanced BAT facility containing twice the number of registers described in the OEA. These additional registers, IBAT4U - IBAT7U, IBAT4L - IBAT7L, DBAT4U - DBAT7U, and DBAT4L - DBAT7L, are available when HID4[SBE] = '1'. In this enhanced mode, up to eight blocks of memory can be mapped for instructions and up to eight blocks of memory can be mapped for data, using the BAT facility.

Implementation Note: The Espresso BAT registers are not initialized by the hardware after the power-up or reset sequence. Consequently, all valid bits in both instruction and data BATs must be cleared before setting any BAT for the first time. If the additional four IBAT and four DBAT register pairs are enabled, or planned to be enabled (by setting HID4[SBE] = '1'), they should also be cleared at the same time. If these additional registers are not enabled (HID4[SBE] = '0'), they need not be cleared. This is true regardless of whether address translation is enabled. Also, software must avoid overlapping blocks while updating a BAT or areas. Even if translation is disabled, multiple BAT hits are treated as programming errors and can corrupt the BAT registers and produce unpredictable results. Always rezero during the reset ISR. After zeroing all BATs, set them (in order) to the desired values. HRESET disorders the BATs. SRESET does not.

5.4 Memory Segment Model

Espresso adheres to the memory segment model as defined in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual for 32-bit implementations. Memory in the PowerPC OEA is divided into 256 MB segments. This segmented memory model provides a way to map 4 KB pages of effective addresses to 4 KB pages in physical memory or 128 KB pages of effective address to 128 KB pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

The segment/page address translation mechanism can be superseded by the block address translation (BAT) mechanism described in *Section 5.3 Block Address Translation*. If not, translation is a 2-step process:

1. Translation from effective address to the virtual address.

Note: The virtual address never exists as a specific entity, but can be considered to be the concatenation of the virtual page number and the byte offset within a page.

2. Translation from virtual address to physical address.

This section highlights those areas of the memory segment model defined by the OEA that are specific to Espresso.

5.4.1 Page History Recording

Referenced (R) and changed (C) bits in each PTE keep history information about the page. They are maintained by a combination of Espresso table search hardware and the system software. The operating system uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Referenced and changed recording is performed only for accesses performed with page address translation and not for translations performed with the BAT mechanism or for accesses that correspond to direct-store ($T = '1'$) segments. Furthermore, R and C bits are maintained only for accesses performed while address translation is enabled ($MSR[IR] = '1'$ or $MSR[DR] = '1'$).

In Espresso, the referenced and changed bits are updated as follows:

- For TLB hits, the C bit is updated according to *Table 5-8*.
- For TLB misses, when a table search operation is in progress to locate a PTE, the R and C bits are updated (set, if required) to reflect the status of the page based on this access.

Table 5-8. Table Search Operations to Update History Bits (TLB Hit Case)

R and C bits in TLB Entry	Processor Action
00	Combination does not occur.
01	Combination does not occur.
10	Read: No special action. Write: Espresso initiates a table search operation to update C.
11	No special action for read or write.

The table shows that the status of the C bit in the TLB entry (in the case of a TLB hit) is what causes the processor to update the C bit in the PTE (the R bit is assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose referenced and changed bits were cleared.

The **dcbt** and **dcbtst** instructions can execute if there is a TLB/BAT hit or if the processor is in real addressing mode. In case of a TLB or BAT miss, these instructions are treated as no-ops; they do not initiate a table search operation and they do not set either the R bit or the C bit.

As defined by the PowerPC Architecture, the referenced and changed bits are updated as if address translation were disabled (real addressing mode). If these update accesses hit in the data cache, they are not seen on the external bus. If they miss in the data cache, they are performed as typical cache line fill accesses on the bus (assuming the data cache is enabled).

5.4.1.1 Referenced Bit

The referenced (R) bit of a page is located in the PTE in the page table. Every time a page is referenced (with a read or write access) and the R bit is zero, Espresso sets the R bit in the page table. The OEA specifies that the referenced bit can be set immediately, or the setting can be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, the referenced bit in all TLB entries is effectively always set. The processor never automatically clears the referenced bit.

The referenced bit is only a hint to the operating system about the activity of a page. At times, the referenced bit can be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this in PowerPC systems follow:

- Fetching of instructions not subsequently executed
- A memory reference caused by a speculatively executed instruction that is mispredicted
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by an **stwcx.** instruction when no store is performed because a reservation does not exist
- Accesses that cause exceptions and are not completed

5.4.1.2 Changed Bit

The changed bit of a page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented, as in Espresso). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the changed bit in the matching TLB entry is checked. If it is already set, it is not updated. If the TLB changed bit is '0', Espresso initiates the table search operation to set the C bit in the corresponding PTE in the page table. Espresso then reloads the TLB (with the C bit set).

The changed bit (in both the TLB and the PTE in the page tables) is set only when a store operation is allowed by the page memory protection mechanism and the store is guaranteed to be in the execution path (unless an exception, other than those caused by the **sc**, **rfi**, or trap instructions, occurs). Furthermore, the following conditions can cause the C bit to be set:

- The execution of an **stwcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of an **stswx** instruction is allowed by the memory protection mechanism but a store operation is not performed because the specified length is zero.
- The store operation is not performed because an exception occurs before the store is performed.

Again, note that although the execution of the **dcbt** and **dcbtst** instructions can cause the R bit to be set, they never cause the C bit to be set.

5.4.1.3 Scenarios for Referenced and Changed Bit Recording

This section provides a summary of the model (defined by the OEA) that is used by PowerPC processors to maintain the referenced and changed bits. In some scenarios, the bits are guaranteed to be set by the processor; in some scenarios, the architecture allows the bits to be set (not absolutely required); and in some scenarios, the bits are guaranteed to not be set. Note that when Espresso updates the R and C bits in memory, the accesses are performed as if MSR[DR] = '0' and G = '0' (that is, as nonguarded cacheable operations in which coherency is required).

Table 5-9 on page 174 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over a matching scenario closer to the bottom of the table. For example, if an **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as a load with respect to address translation. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by the cache management instructions that are treated as a store with respect to address translation.

Table 5-9. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit		Causes Setting of C Bit	
		OEA	Espresso	OEA	Espresso
1	No-execute protection violation	No	No	No	No
2	Page protection violation	Maybe	Yes	No	No
3	Out-of-order instruction fetch or load operation	Maybe	No	No	No
4	Out-of-order store operation. Would be required by the sequential execution model in the absence of system-caused or imprecise exceptions, or of floating-point assist exception for instructions that would cause no other kind of precise exception.	Maybe ¹	No	No	No
5	All other out-of-order store operations	Maybe ¹	No	Maybe ¹	No
6	Zero-length load (lswx)	Maybe	No	No	No
7	Zero-length store (stswx)	Maybe ¹	No	Maybe ¹	No
8	Store conditional (stwcx.) that does not store	Maybe ¹	Yes	Maybe ¹	Yes
9	In-order instruction fetch	Yes	Yes	No	No
10	Load instruction or eciwx	Yes	Yes	No	No
11	Store instruction, ecowx , dcbz_l , or dcbz instruction	Yes	Yes	Yes	Yes
12	icbi , dcbt , or dcbtst instruction	Maybe	No	No	No
13	dcbst or dcbf instruction	Maybe	Yes	No	No
14	dcbi instruction	Maybe ¹	Yes	Maybe ¹	Yes

1. If C is set, R is guaranteed to be set also.

For more information, see the page history recording section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.4.2 Page Memory Protection

Espresso implements page memory protection as it is defined in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

5.4.3 TLB Description

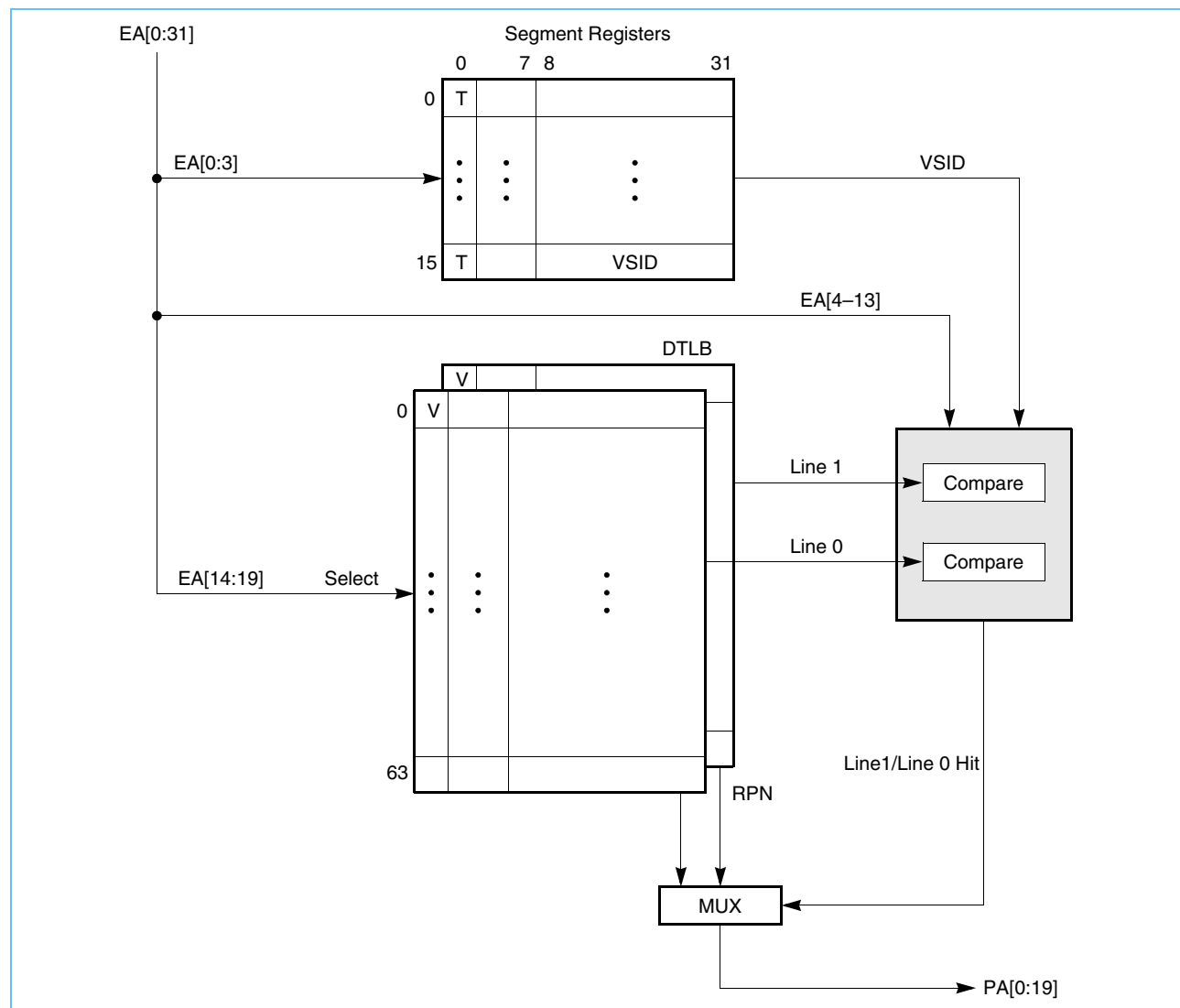
Espresso implements separate 128-entry data and instruction TLBs to maximize performance. This section describes the hardware resources provided in Espresso to facilitate page address translation. Note that the hardware implementation of the MMU is not specified by the architecture. This description applies to Espresso, but it does not necessarily apply to other PowerPC processors.

5.4.3.1 TLB Organization

Because Espresso has two MMUs (IMMU and DMMU) that operate in parallel, some of the MMU resources are shared, and some are actually duplicated (shadowed) in each MMU to maximize performance. For example, although the architecture defines a single set of segment registers for the MMU, Espresso maintains two identical sets of segment registers, one for the IMMU and one for the DMMU. When an instruction that updates the segment register executes, Espresso automatically updates both sets.

Each TLB contains 128 entries organized as a 2-way, set-associative array with 64 sets. *Figure 5-7* shows the DTLB organization. The ITLB organization is the same. When an address is being translated, a set of two TLB entries is indexed in parallel with the access to a segment register. If the address in one of the two TLB entries is valid and matches the 40-bit virtual page number, that TLB entry contains the translation. If no match is found, a TLB miss occurs. *Figure 5-7* assumes the use of 4 KB pages. In large-page mode, EA[9:14] select the TLB set, EA[4:8] go directly to the compare logic, and the translation provides bits 0:14 of the physical address.

Figure 5-7. Segment Register and DTLB Organization



Unless the access is the result of an out-of-order access, a hardware table search operation begins if there is a TLB miss. If the access is out of order, the table search operation is postponed until the access is required, at which point the access is no longer out of order. When the matching PTE is found in memory, it is loaded into the TLB entry selected by the least-recently-used (LRU) replacement algorithm, and the translation process begins again, this time with a TLB hit.

To uniquely identify a TLB entry as the required PTE, the TLB entry also contains 4 more bits of the page index, EA[10:13] (in addition to the abbreviated page index [API] bits in the PTE).

Software cannot access the TLB arrays directly, except to invalidate an entry with the **tlbie** instruction.

Each set of TLB entries has one associated LRU bit. The LRU bit for a set is updated any time either entry is used, even if the access is speculative. Invalid entries are always the first to be replaced.

Although both MMUs can be accessed simultaneously (both sets of segment registers and TLBs can be accessed in the same clock), only one exception condition can be reported at a time. ITLB miss exception conditions are reported when there are no more instructions to be dispatched or retired (the pipeline is empty), and DTLB miss exception conditions are reported when the load or store instruction is ready to be retired. See *Section 6 Instruction Timing* on page 183 for more detailed information about the internal pipelines and the reporting of exceptions.

When an instruction or data access occurs, the effective address is routed to the appropriate MMU. EA[0:3] select one of the 16 segment registers, and the remaining effective address bits and the VSID field from the segment register are passed to the TLB. EA[14:19] (EA[9:14] for large pages) then selects two entries in the TLB. The valid bits are checked, and the 40-bit virtual page number (24-bit VSID and EA[4:19]) must match the VSID, effective address page index (EAPI), and API fields of the TLB entries. If one of the entries hits, the page protection (PP) bits are checked for a protection violation. If these bits do not cause an exception, the C bit is checked and a table search operation is initiated if C must be updated. If C does not require updating, the RPN value is passed to the memory subsystem, and the WIMG bits are then used as attributes for the access.

Although address translation is disabled on a reset condition, the valid bits of TLB entries are not automatically cleared. Thus, TLB entries must be explicitly cleared by the system software (with the **tlbie** instruction) before the valid entries are loaded and address translation is enabled. Also, note that the segment registers do not have a valid bit, and so they should also be initialized before translation is enabled.

5.4.3.2 TLB Invalidation

Espresso implements the optional **tlbie** and **tlbsync** instructions, which are used to invalidate TLB entries. The execution of the **tlbie** instruction always invalidates four entries: each of the two ITLB and the two DTLB entries indexed by EA[14:19] (EA[9:14] for large pages).

The architecture allows **tlbie** to optionally enable a TLB invalidate signaling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. Espresso broadcasts the TLB invalidation to other processors and snoops TLB invalidation operations from other processors.

Note: The architecture specifies that a **tlbie** instruction must be issued on only one processor at a time in a multiprocessor system. The architecture also specifies that a barrier instruction (**ieieio** or **sync**) is required between a **tlbie** instruction and a **tlbsync** instruction. In this implementation, a **sync** followed by an **isync** is required between a **tlbie** instruction and a **tlbsync** instruction to ensure that the **tlbsync** instruction is executed after the other cores have had a chance to snoop the TLBIE transaction. Finally, the architecture spec-

ifies that a **sync** instruction should follow the **tlbsync** instruction to ensure that the **tlbsync** completes before another core executes a **tlbie** or **tlbsync** instruction.

The **tlbsync** instruction causes instruction execution to stop if there are snooped TLB invalidation operations pending in other cores. Each core asserts an internal **tlbisync** signal to the other cores when it has such a pending TLB invalidation operation. The **tlbsync** instruction waits to be completed until these signals are negated.

The **tlbia** instruction is not implemented on Espresso; when its opcode is encountered, an illegal instruction program exception is generated. To invalidate all entries of both TLBs, 64 **tlbie** instructions must be executed, incrementing the value in EA[14:19] (EA[9:14] for large pages) by one each time. See the instruction set section of the *PowerPC Microprocessor Family: The Programming Environments* manual for detailed information about this instruction.

Software must ensure that instruction fetches or memory references to the virtual pages specified by the **tlbie** have been completed before executing the **tlbie** instruction.

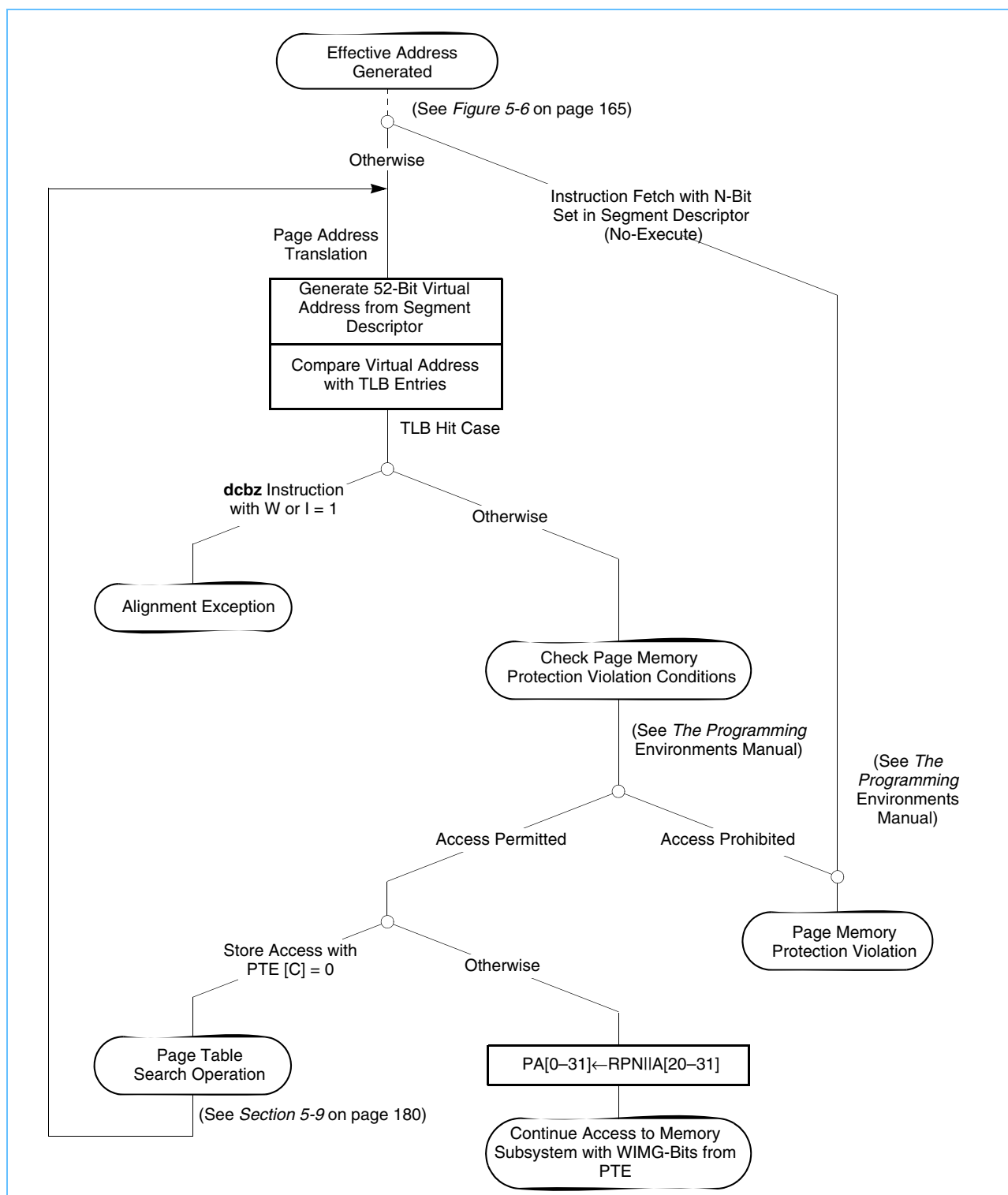
Other than the possible TLB miss on the next instruction prefetch, the **tlbie** instruction does not affect the instruction fetch operation; that is, the prefetch buffer is not purged and does not cause these instructions to be refetched.

5.4.4 Page Address Translation Summary

Figure 5-8 on page 178 provides the detailed flow for the 4 KB page address translation mechanism. The flow is the same for large page translation, but results in a physical address comprised of the 15-bit RPN concatenated with the 17-bit page offset. The figure includes the checking of the N bit in the segment descriptor and then expands on the TLB Hit branch of *Figure 5-6* on page 165. The detailed flow for the TLB Miss branch of *Figure 5-6* is described in *Section 5.4.5 Page Table Search Operation* on page 179.

Note: As in the case of block address translation, if an attempt is made to execute a **dcbz** or **dcbz_l** instruction to a page marked either write-through or caching-inhibited ($W = '1'$ or $I = '1'$), an alignment exception is generated. The checking of memory protection violation conditions is described in the memory management section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Figure 5-8. Page Address Translation Flow (TLB Hit)



5.4.5 Page Table Search Operation

If the translation is not found in the TLBs (a TLB miss), Espresso initiates the table search operation. See the PTE format for 32-bit implementations section of the *PowerPC Microprocessor Family: The Programming Environments* manual for PTE formats. For large pages, RPN[0:14] in the PTE is used for the real page number, while RPN[15:19] is unused.

The following is a summary of the page table search process performed by Espresso:

1. The 32-bit physical address of the primary PTEG is generated as described in the page table addresses section of the *PowerPC Microprocessor Family: The Programming Environments* manual. See *Section 5.1.4 Large Page Support* on page 160 for the hash function used for large-page mode.
2. The first PTE (PTE0) in the primary PTEG is read from memory. PTE reads occur with an implied WIM memory/cache mode control bit setting of '001'. Therefore, they are considered cacheable and are read (burst) from memory and placed in the cache.
3. The PTE in the selected PTEG is tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
 - PTE[H] = '0'
 - PTE[V] = '1'
 - PTE[VSID] = VA[0:23]
 - PTE[API] = VA[24:29]
4. If a match is not found, step 3 is repeated for each of the other seven PTEs in the primary PTEG. If a match is found, the table search process continues as described in step 8. If a match is not found within the eight PTEs of the primary PTEG, the address of the secondary PTEG is generated.
5. The first PTE (PTE0) in the secondary PTEG is read from memory. Again, because PTE reads have a WIM bit combination of '001', an entire cache line is read into the on-chip cache.
6. The PTE in the selected secondary PTEG is tested for a match with the virtual page number (VPN) of the access. For a match to occur, the following must be true:
 - PTE[H] = '1'
 - PTE[V] = '1'
 - PTE[VSID] = VA[0:23]
 - PTE[API] = VA[24:29]
7. If a match is not found, step 6 is repeated for each of the other seven PTEs in the secondary PTEG. If it is never found, an exception is taken (step 9).
8. If a match is found, the PTE is written into the on-chip TLB and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if the access is a write operation) and the table search is complete.
9. If a match is not found within the eight PTEs of the secondary PTEG, the search fails, and a page fault exception condition occurs (either an ISI exception or a DSI exception).

Figure 5-9 on page 180 and *Figure 5-10* on page 181 show how the conceptual model for the primary and secondary page table search operations, described in the *PowerPC Microprocessor Family: The Programming Environments* manual, are realized in Espresso.

Figure 5-9 shows the case of a **dcbz** or **dcbz_l** instruction that is executed with W = '1' or I = '1'; the R bit can be updated in memory (if required) before the operation is performed or the alignment exception occurs. The R bit can also be updated if memory protection is violated.

Figure 5-9. Primary Page Table Search

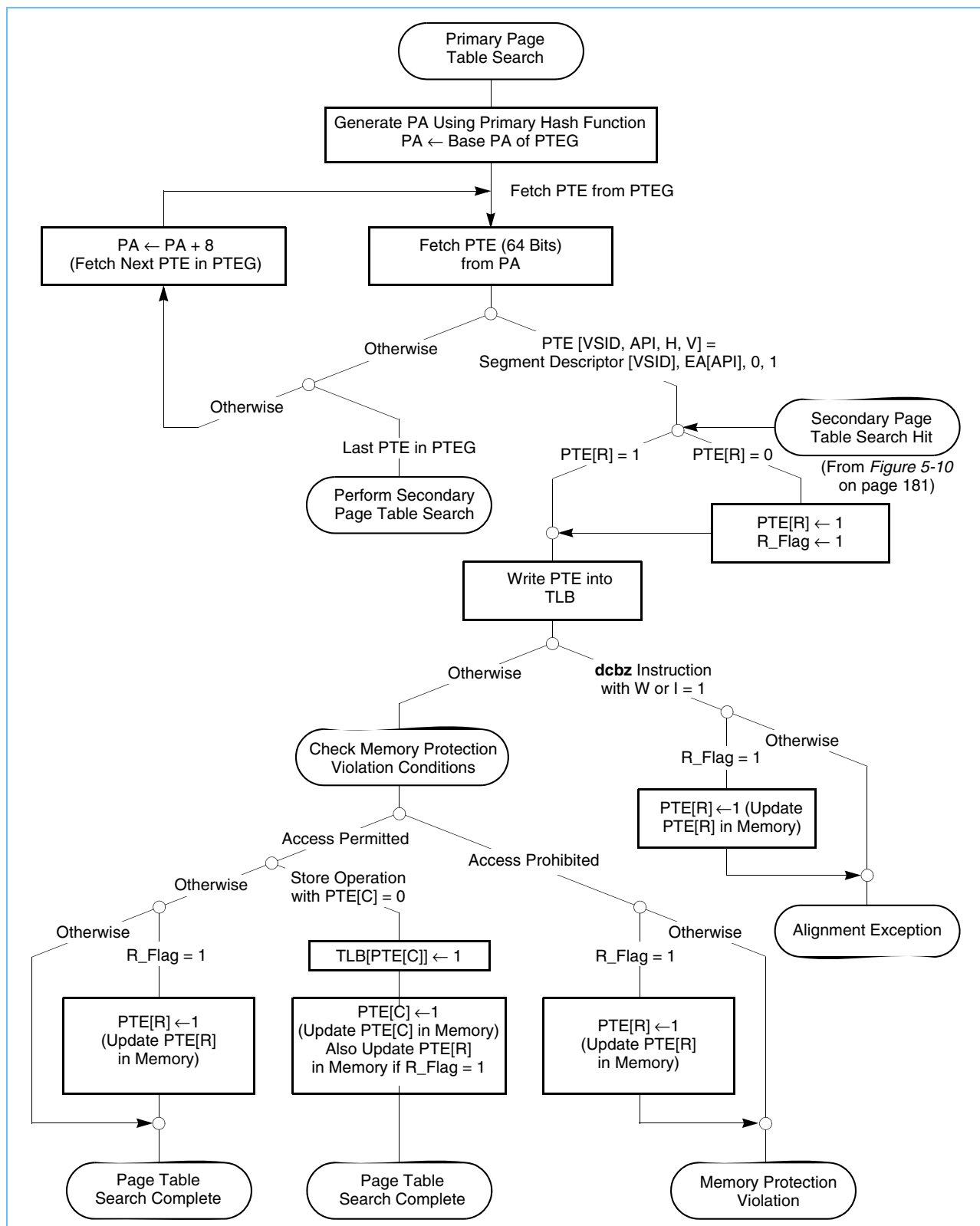
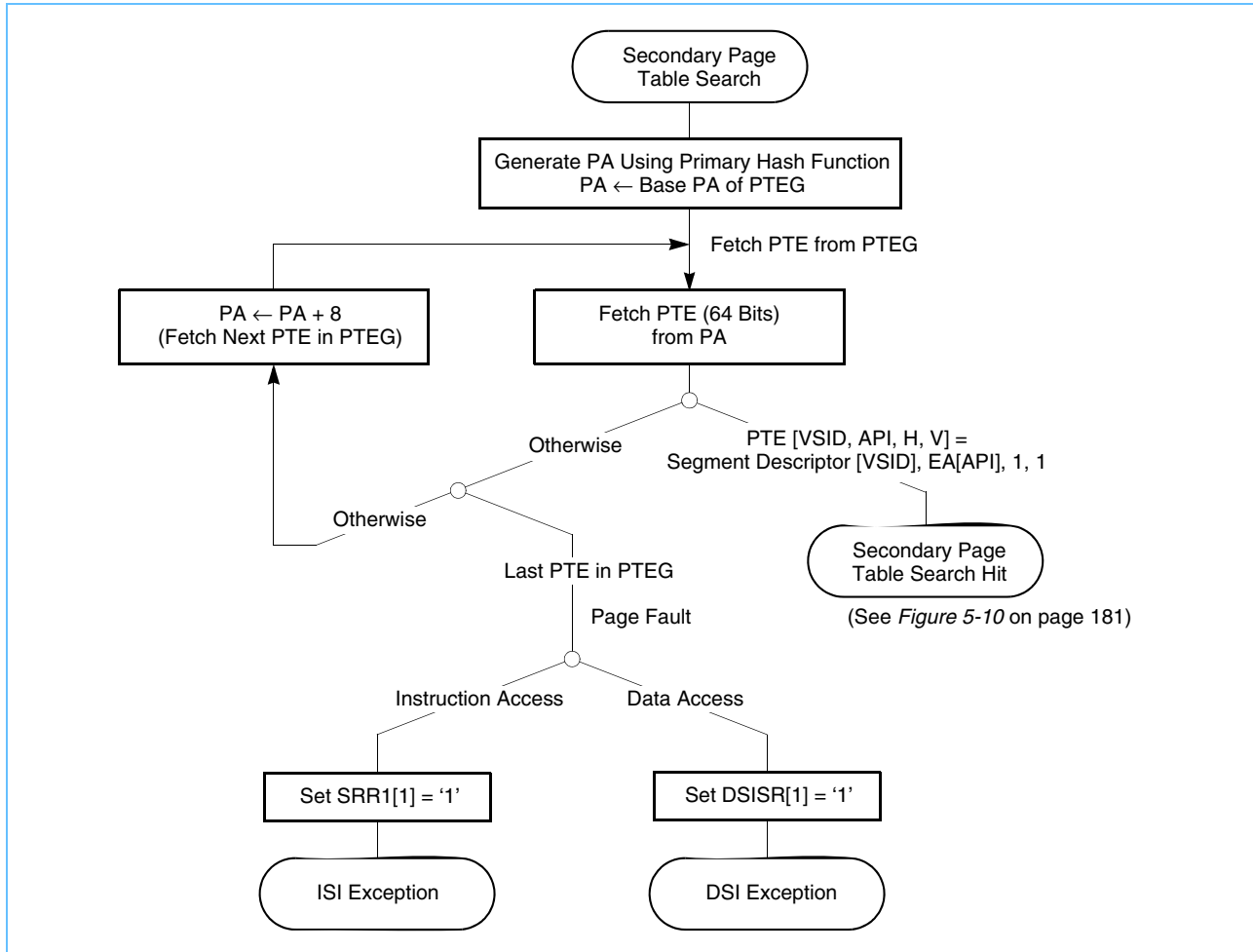


Figure 5-10. Secondary Page Table Search Flow



The LSU initiates out-of-order accesses without knowledge of whether it is legal to do so. Therefore, the MMU does not perform a hardware table search due to TLB misses until the request is required by the program flow. In these out-of-order cases, the MMU does detect protection violations and whether a **dcbz** or **dcbz_l** instruction specifies a page marked as write-through or cache-inhibited. The MMU also detects alignment exceptions caused by the **dcbz** or **dcbz_l** instruction and prevents the changed bit in the PTE from being updated erroneously in these cases.

If an MMU register is being accessed by an instruction in the instruction stream, the IMMU stalls for one translation cycle to perform that operation. The sequencer serializes instructions to ensure the data correctness. For updating the IBATs and SRs, the sequencer classifies those operations as fetch serializing. After such an instruction is dispatched, the instruction buffer is flushed and the fetch stalls until the instruction completes. However, for reading from the IBATs, the operation is classified as execution serializing. As long as the LSU ensures that all previous instructions can be executed, subsequent instructions can be fetched and dispatched.

5.4.6 Page Table Updates

When TLBs are implemented (as in Espresso), they are defined as noncoherent caches of the page tables. TLB entries must be flushed explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified.

Processors can write referenced and changed bits with unsynchronized, atomic byte store operations. Note that the V, R, and C bits each reside in a distinct byte of a PTE. Therefore, extreme care must be taken to use byte writes when updating only one of these bits.

Explicitly altering certain MSR bits (using the **mtmsr** instruction), or explicitly altering PTEs, or certain system registers, can have the side effect of changing the effective or physical addresses from which the current instruction stream is being fetched. This kind of side effect is defined as an implicit branch. Implicit branches are not supported and an attempt to perform one causes boundedly-undefined results. Therefore, PTEs must not be changed in a manner that causes an implicit branch.

The PowerPC register set section of the *PowerPC Microprocessor Family: The Programming Environments* manual lists the possible implicit branch conditions that can occur when system registers and MSR bits are changed.

5.4.7 Segment Register Updates

Synchronization requirements for using the Move to Segment Register instructions are described in the synchronization requirements for special registers and for lookaside buffers section of the *PowerPC Microprocessor Family: The Programming Environments* manual. In particular, note that a context synchronizing instruction, such as an **isync**, must be placed between any **mtsr** or **mtsrin** instruction and a subsequent instruction that causes data translation to take place using any of the segment registers.

6. Instruction Timing

This section describes how the Espresso core fetches, dispatches, and executes instructions and how it reports the results of instruction execution. It gives detailed descriptions of how Espresso execution units work, and how those units interact with other parts of the processor, such as the instruction fetching mechanism, register files, and caches. It gives examples of instruction sequences, showing potential bottlenecks and how to minimize their effects. Finally, it includes tables that identify the unit that executes each instruction implemented on Espresso, the latency for each instruction, and other information that is useful for the assembly language programmer.

Note: The results and conclusions from this section refer to only one of the Espresso cores.

6.1 Terminology and Conventions

This section provides an alphabetical glossary of terms used. These definitions are provided as a review of commonly used terms and as a way to point out specific ways these terms are used in this section.

- **Branch prediction:** The process of guessing whether a branch will or will not be taken. Such predictions can be correct or incorrect; the term “predicted” as it is used here does not imply that the prediction is correct (successful). Instructions along the predicted path are fetched and dispatched to their respective execution units conditionally and can reach the completion unit. However, these instructions must first be validated by the branch resolution process before they can be retired. The PowerPC Architecture defines a means for static branch prediction as part of the instruction encoding. The Espresso core implements two types of dynamic branch prediction. See *Section 6.4.1.2 Branch Instructions and Completion* on page 201.
- **Branch resolution:** The determination of the path that a branch instruction must take. If a branch prediction and branch resolution occur on the same cycle, the processor simply fetches instructions on the correct path as determined by the branch instruction. For predicted branches, branch resolution must determine if the prediction was correct. If the prediction was correct, all speculatively fetched instructions that have been passed to their execution units are validated. If the prediction was wrong, the speculatively fetched instructions must be invalidated (flushed) and instruction fetching must resume along the other path for the branch instruction.
- **Completion:** Completion occurs when an instruction has finished executing and its results are stored in a rename register that had been allocated to it by the dispatch unit. These results are available to subsequent instructions or previously predicted branches.
- **Dispatch:** The process of moving an instruction from the instruction queue to an execution unit. In the Espresso core, the dispatch unit can process up to three instruction in a single cycle if one of the three is a branch. For the nonbranch type instructions, the dispatch unit must do a partial decode to determine the type of instruction to pass it to its respective execution unit. Also, a rename register and a place in the completion queue must be reserved; otherwise, a stall occurs. If a branch updates either the LR or CTR Register, it also must be allocated to a completion queue entry.
- **Fall-through:** A not-taken branch.
- **Fetch:** The process of bringing instructions from the system memory (such as a cache or the main memory) into the instruction queue.
- **Folding (branch folding):** In Espresso, a branch is expunged from (folded out of) the instruction queue by using the dispatch mechanism, without either being passed to an execution unit and or given a position in the completion queue. Subsequent instructions are fetched from the target address calculated by the

branch instruction, for branches taken or sequential instructions following the branch for a branch-not-taken, and placed into a reservation register to which the instruction is dispatched.

- **Finish:** Finishing occurs in the last cycle of execution. (This can also be the first cycle of execution for instructions that only require one cycle for execution.) In this cycle, the output rename register and the completion queue entry are updated to indicate that the instruction has finished executing.
- **Latency:** The number of clock cycles necessary to execute an instruction and make ready the results of that execution for a subsequent instruction.
- **Pipeline:** In the context of instruction timing, the term “pipeline” refers to the interconnection of the stages. The events necessary to process an instruction are broken into several cycle-length tasks to allow work to be performed on several instructions simultaneously; this is analogous to an assembly line. As an instruction is processed, it passes from one stage to the next. When it does, the stage becomes available for the next instruction.
- Although an individual instruction can take many cycles to complete (the number of cycles is called instruction latency), pipelining makes it possible to overlap the processing so that the throughput (number of instructions completed per cycle) is greater than if pipelining were not implemented.
- **Program order:** The order of instructions in an executing program. More specifically, this term is used to refer to the original order in which program instructions are fetched into the instruction queue from the system memory.
- **Rename register:** Temporary buffers used to hold either source or destination values for instructions that are in a stage of execution. This simplifies the passing of data outside of the general purpose register file (GPR) between instructions during execution.
- **Reservation station:** A buffer between the dispatch and execution units where instructions await execution.
- **Retirement:** Removal of a completed instruction from the completion queue. At this time, any output from the completed instruction is written to the appropriate architected destination register. This might be a GPR, FPR, or a CR field.
- **Stage:** The processing of instructions in Espresso is done in stages. They are: fetch, decode/dispatch, execute, complete, and retirement. The fetch unit brings instructions from the memory system into the instruction queue. Once in the instruction queue, the dispatch unit must do a partial decode on the instruction to determine its type. If the instruction is an integer, it is passed to the integer execution unit; if it is a floating-point type, it is passed to the floating-point execution unit; if it is a branch it is processed immediately by branch folding and branch prediction functions. Instructions spend one or more cycles in each stage as they are being processed by the Espresso core.
- **Stall:** An occurrence when an instruction cannot proceed to the next stage. An instruction can spend multiple cycles in one stage. An integer multiply, for example, takes multiple cycles in the execute stage. When this occurs, subsequent instructions can stall.
- **Superscalar:** A superscalar processor is one that has multiple execution units. The Espresso core has one floating-point unit, two integer units, one load/store unit, and a system unit for miscellaneous instructions. PowerPC instructions are processed in parallel by these execution units.
- **Throughput:** A measure of the total number of instructions that are processed by all execution units per unit of time.
- **Write-back:** Write-back (in the context of instruction handling) occurs when a result is written into the architectural registers (typically the GPRs and FPRs). Results are written back at retirement time from rename registers for most instructions. The instruction is also removed from the completion queue at this time.

6.2 Instruction Timing Overview

The Espresso design minimizes average instruction execution latency: the number of clock cycles it takes to fetch, decode, dispatch, and execute instructions and make the results available for a subsequent instruction. Some instructions, such as loads and stores, access memory and require additional clock cycles between the execute phase and the write-back phase. These latencies vary depending on the following conditions:

- Whether the access is to cacheable or noncacheable memory
- Whether it hits in the L1 or L2 cache
- Whether the cache access generates a write-back to memory
- Whether the access causes a snoop hit from another device that generates additional activity
- Other conditions that affect memory accesses

The Espresso core implements many features to improve throughput, such as pipelining, superscalar instruction issue, branch folding, 2-level speculative handling, two types of branch prediction, and multiple execution units that operate independently and in parallel.

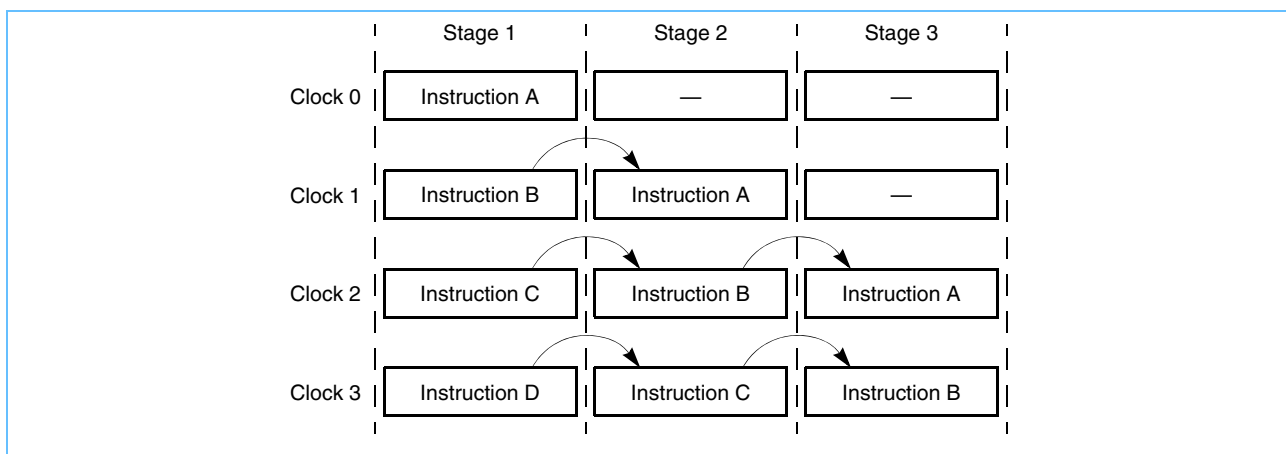
As an instruction passes from stage to stage in a pipelined system, multiple instructions are in various stages of execution at any given time. Also, with multiple execution units operating in parallel, more than one instruction can be completed in a single cycle.

The Espresso core contains the following execution units that operate independently and in parallel:

- Branch processing unit (BPU)
- Integer unit 1 (IU1): executes all integer instructions
- Integer unit 2 (IU2): executes all integer instructions except multiplies and divides
- 64-bit floating-point unit (FPU)
- Load/store unit (LSU)
- System register unit (SRU)

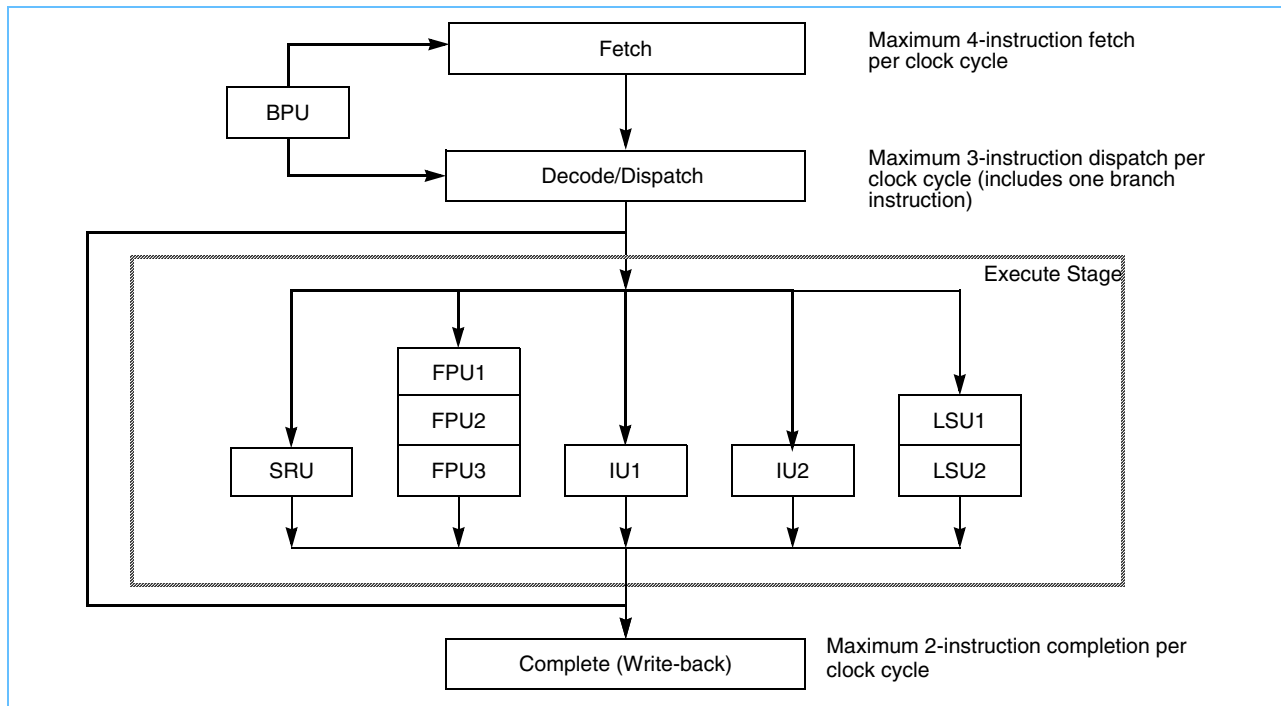
Figure 6-1 represents a generic pipelined execution unit.

Figure 6-1. Pipelined Execution Unit



Each Espresso core can retire two instructions on every clock cycle. In general, Espresso processes instructions in four stages—fetch, decode/dispatch, execute, and complete as shown in *Figure 6-2* on page 186. Note that the example of a pipelined execution unit in *Figure 6-1* is similar to the three-stage FPU pipeline in *Figure 6-2*.

Figure 6-2. Superscalar/Pipeline Diagram



The instruction pipeline stages are described as follows:

- The instruction fetch stage includes the clock cycles necessary to request instructions from the memory system and the time the memory system takes to respond to the request. Instruction fetch timing depends on many variables, such as whether the instruction is in the branch target instruction cache, the L1 instruction cache, or the L2 cache. Additional factors affect instruction fetch timing when it is necessary to fetch instructions from system memory; those factors include the processor-to-bus clock ratio, the amount of bus traffic, and whether any cache coherency operations are required.

Because there are so many variables, unless otherwise specified, the instruction timing examples below assume optimal performance, and that the instructions are available in the instruction queue in the same clock cycle that they are requested. The fetch stage ends when the instruction is dispatched.

- The decode/dispatch stage consists of the time it takes to decode the instruction and dispatch it from the instruction queue to the appropriate execution unit. Instruction dispatch requires the following:
 - Instructions can be dispatched only from the two lowest instruction queue entries, IQ0 and IQ1.
 - A maximum of two instructions can be dispatched per clock cycle (and one additional branch instruction can be handled by the BPU).
 - Only one instruction can be dispatched to each execution unit per clock cycle.
 - There must be a vacancy in the specified execution unit reservation station.
 - A rename register must be available for each destination operand specified by the instruction.

- For an instruction to dispatch, the appropriate execution unit reservation station must be available and there must be an open position in the completion queue. If no entry is available, the instruction remains in the IQ.
- The execute stage consists of the time between dispatch to the execution unit (or reservation station) and the point at which the instruction vacates the execution unit.

Most integer instructions have a one-cycle latency; results of these instructions can be used in the clock cycle after an instruction enters the execution unit. However, integer multiply and divide instructions take multiple clock cycles to complete. The IU1 can process all integer instructions; the IU2 can process all integer instructions except multiply and divide instructions.

The LSU and FPU are pipelined (as shown in *Figure 6-2* on page 186).

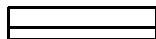
- The complete (complete/write-back) pipeline stage maintains the correct architectural machine state and commits it to the architectural registers at the proper time. If the completion logic detects an instruction containing an exception status, all following instructions are cancelled, their execution results in rename registers are discarded, and the correct instruction stream is fetched.

The complete stage ends when the instruction is retired. Two instructions can be retired per cycle. Instructions are retired only from the two lowest completion queue entries, CQ0 and CQ1.

The notation conventions used in the instruction timing examples are as follows:



Fetch. The fetch stage includes the time between when an instruction is requested and when it is brought into the instruction queue. This latency can be vary, depending upon whether the instruction is in the BTIC, the L1 cache, the L2 cache, or system memory (in which case latency can be affected by bus speed and traffic on the system bus, and address translation issues). Therefore, in the examples in this section, the fetch stage is usually idealized, that is, an instruction is usually shown to be in the fetch stage when it is a valid instruction in the instruction queue. The instruction queue has six entries, IQ0 - IQ5.



In dispatch entry (IQ0/IQ1). Instructions can be dispatched from IQ0 and IQ1. Because dispatch is instantaneous, it is perhaps more useful to describe it as an event that marks the point in time between the last cycle in the fetch stage and the first cycle in the execute stage.



Execute. The operations specified by an instruction are being performed by the appropriate execution unit. The black stripe is a reminder that the instruction occupies an entry in the completion queue, described in *Figure 6-3* on page 188.



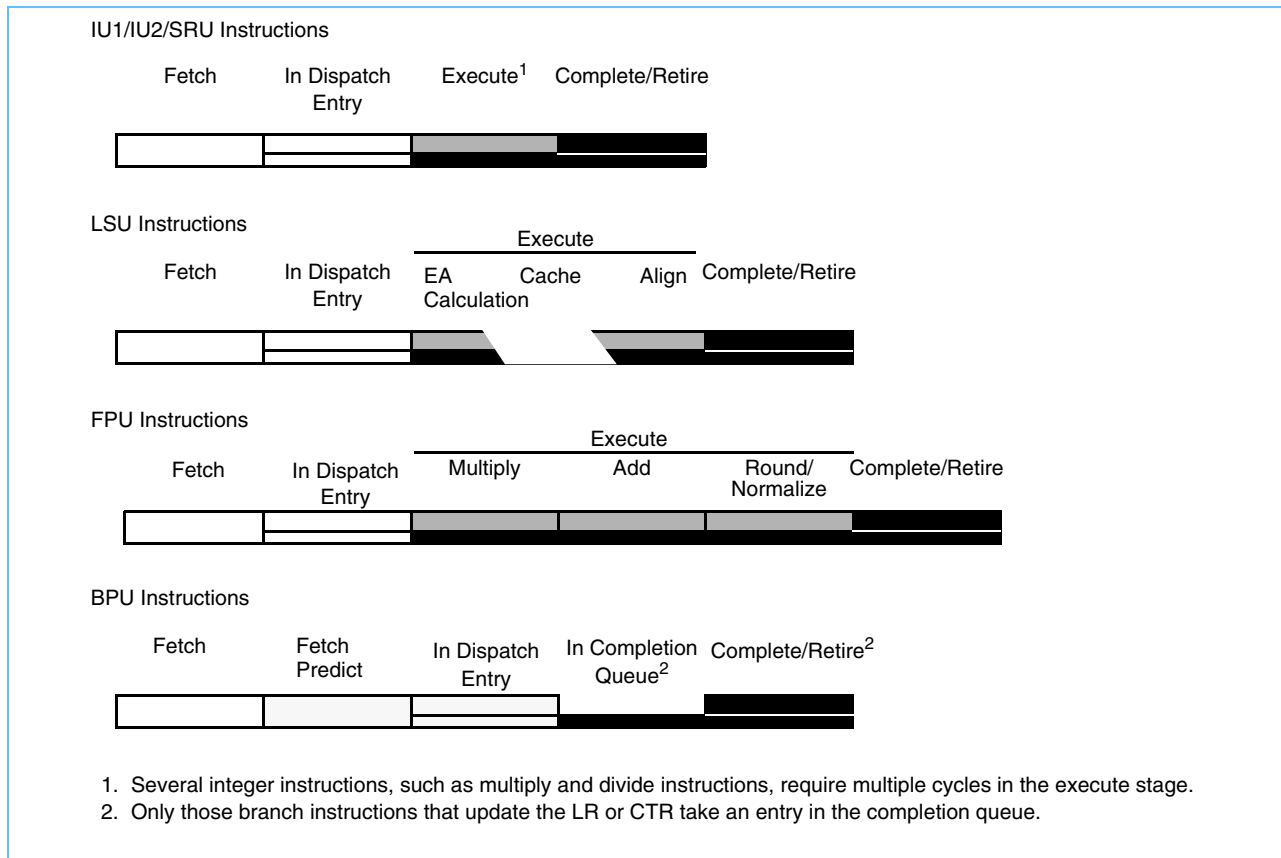
Complete. The instruction is in the completion queue. In the final stage, the results of the executed instruction are written back and the instruction is retired. The completion queue has six entries, CQ0 - CQ5.



In retirement entry. Completed instructions can be retired from CQ0 and CQ1. Like dispatch, retirement is an event that in this case occurs at the end of the final cycle of the complete stage.

Figure 6-3 on page 188 shows the stages of the Espresso execution units.

Figure 6-3. Espresso Microprocessor Pipeline Stages



6.3 Timing Considerations

Espresso is a superscalar processor; as many as three instructions can be issued to the execution units (one branch instruction to the branch processing unit, and two instructions issued from the dispatch queue to the other execution units) during each clock cycle. Only one instruction can be dispatched to each execution unit.

Although instructions appear to the programmer to execute in program order, the Espresso core improves performance by executing multiple instructions at a time, using hardware to manage dependencies. When an instruction is dispatched, the register file or rename register from a previous instruction provides the source data to the execution unit. The register files and rename register have sufficient bandwidth to allow dispatch of two instructions per clock under most conditions.

The Espresso BPU decodes and executes branches immediately after they are fetched. When a conditional branch cannot be resolved due to a CR data (or any) dependency, the branch direction is predicted and execution continues on the predicted path. If the prediction is incorrect, the following steps are taken:

1. The instruction queue is purged and fetching continues from the correct path.
2. Any instructions behind (in program order) the predicted branch in the completion queue are allowed to complete.

3. Instructions fetched on the mispredicted path of the branch are purged.
4. Fetching resumes along the correct (other) path.

After an execution unit finishes executing an instruction, it places resulting data into the appropriate GPR or FPR rename register. The results are then stored into the correct GPR or FPR during the write-back stage (retirement). If a subsequent instruction needs the result as a source operand, it is made available simultaneously to the appropriate execution unit, which allows a data-dependent instruction to be decoded and dispatched without waiting to read the data from the register file. Branch instructions that update either the LR or CTR write back their results in a similar fashion.

The following section describes this process in greater detail.

6.3.1 General Instruction Flow

As many as four instructions can be fetched into the instruction queue (IQ) in a single clock cycle. Instructions enter the IQ and are issued to the various execution units from the dispatch queue. The Espresso core tries to keep the IQ full at all times, unless instruction cache throttling is operating.

The number of instructions requested in a clock cycle is determined by the number of vacant spaces in the IQ during the previous clock cycle. This is shown in the examples in this chapter. Although the instruction queue can accept as many as four new instructions in a single clock cycle, if only one IQ entry is vacant, only one instruction is fetched. Typically instructions are fetched from the L1 instruction cache, but they can also be fetched from the branch target instruction cache (BTIC) if a branch is taken. If the branch taken instruction request hits in the BTIC, it can usually present the first two instructions of the new instruction stream in the next clock cycle, giving enough time for the next pair of instructions to be fetched from the instruction L1 cache resulting in no idle cycles in the instruction stream (also known as the zero cycle branch). If instructions are not in the BTIC or the L1 instruction cache, they are fetched from the L2 cache or from system memory.

The Espresso instruction cache throttling feature, managed through the instruction cache throttling control (ICTC) register, can lower the processor overall junction temperature by slowing the instruction fetch rate.

Branch instructions are identified by the fetcher, and forwarded to the BPU directly, bypassing the dispatch queue. If the branch is unconditional or if the specified conditions are already known, the branch can be resolved immediately. That is, the branch direction is known and instruction fetching can continue along the correct path. Otherwise, the branch direction must be predicted.

Espresso offers several resources to aid in quick resolution of branch instructions and to improve the accuracy of branch predictions. These include the following:

- Branch target instruction cache: The 64-entry (4-way, set-associative) branch target instruction cache (BTIC) holds branch target instructions. When a branch is encountered in a repeated loop, usually the first two instructions in the target stream can be fetched into the instruction queue on the next clock cycle. The BTIC can be disabled and invalidated through bits in the HID0 Register. Coherency of the BTIC table is maintained by table reset on an instruction cache (I-Cache) flash invalidate, **icbi**, **isync**, or **rfi** instruction execution, or when an exception is taken.
- Dynamic branch prediction: The 512-entry branch history table (BHT) is implemented with 2 bits per entry for four degrees of prediction: not-taken, strongly not-taken, taken, strongly taken. Whether a branch instruction is taken or not-taken can change the strength of the next prediction. This dynamic branch prediction is not defined by the PowerPC Architecture.

To reduce aliasing, only predicted branches update the BHT entries. Dynamic branch prediction is enabled by setting HID0[BHT]; otherwise, static branch prediction is used.

- **Static branch prediction:** Static branch prediction is defined by the PowerPC Architecture and involves encoding the branch instructions. See *Static Branch Prediction* on page 202.

Branch instructions that do not update the LR or CTR are removed from the instruction stream by branch folding, as described in *Section 6.4.1.1* on page 199. Branch instructions that update the LR or CTR are treated as if they require dispatch (even though they are not issued to an execution unit in the process). They are assigned a position in the completion queue to ensure that the CTR and LR are updated in correct program order.

All other instructions are issued from IQ0 and IQ1. The dispatch rate depends upon the availability of resources such as the execution units, rename registers, and completion queue entries, and upon the serializing behavior of some instructions. Instructions are dispatched in program order; an instruction in IQ1 cannot be dispatched ahead of one in IQ0.

6.3.2 Instruction Fetch Timing

Instruction fetch latency depends on whether the fetch hits the BTIC, the L1 instruction cache, or the L2 cache. If no cache hit occurs, a memory transaction is required in which case fetch latency is affected by bus traffic, bus clock speed, and memory translation. These issues are discussed in the following sections.

6.3.2.1 Cache Arbitration

When the instruction fetcher requests instructions from the instruction cache, two things can happen. If the instruction cache is idle and the requested instructions are present, they are provided on the next clock cycle. However, if the instruction cache is busy due to a cache-line-reload operation, instructions cannot be fetched until that operation completes.

6.3.2.2 Cache Hit

If the instruction fetch hits the instruction cache, it takes only one clock cycle after the request for as many as four instructions to enter the instruction queue. Note that the cache is not blocked to internal accesses during a cache reload completes (hits under misses). The critical doubleword is written simultaneously to the cache and forwarded to the requesting unit, minimizing stalls due to load delays.

Figure 6-4 shows the paths taken by instructions.

Figure 6-4. Instruction Flow Diagram

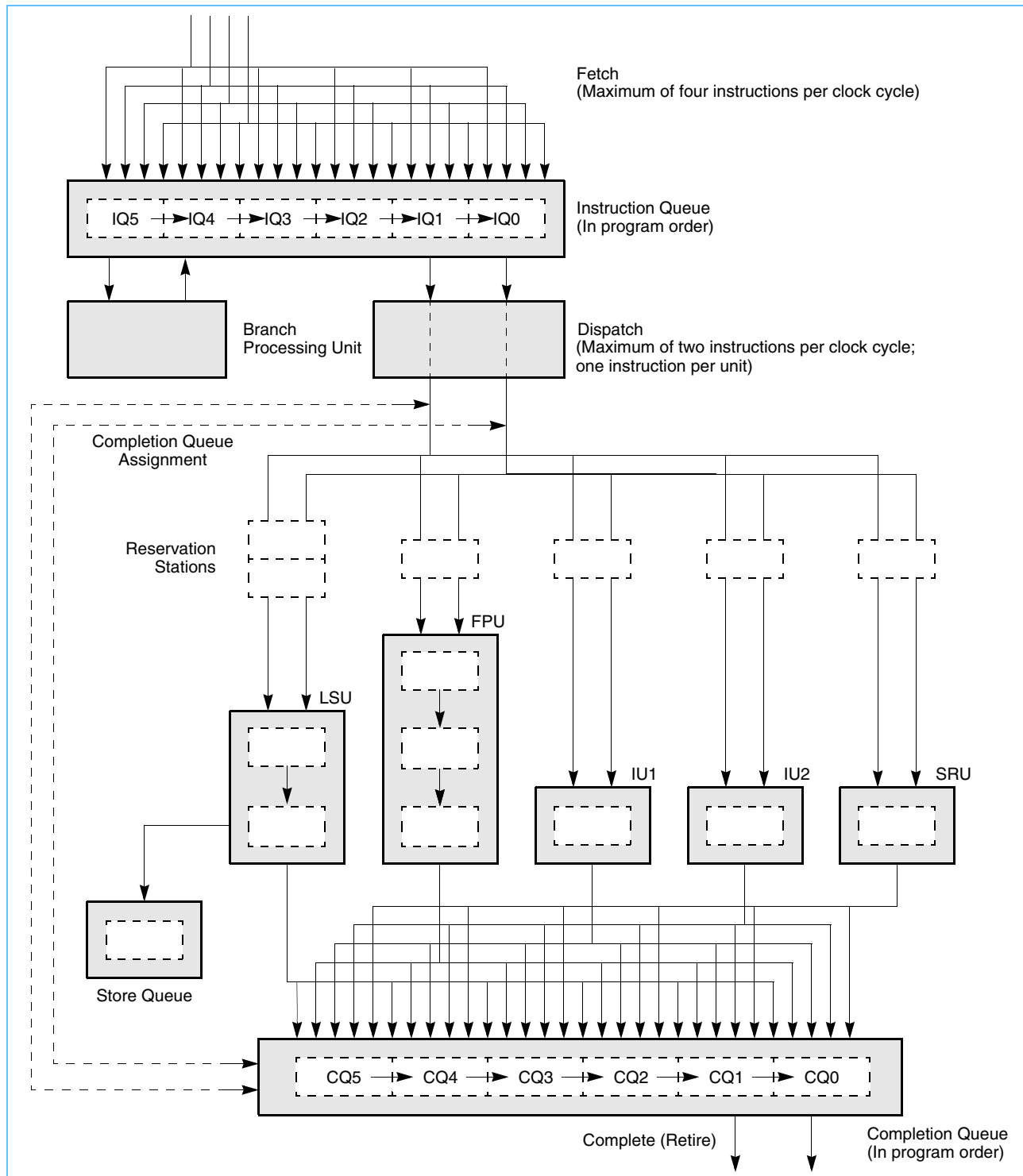
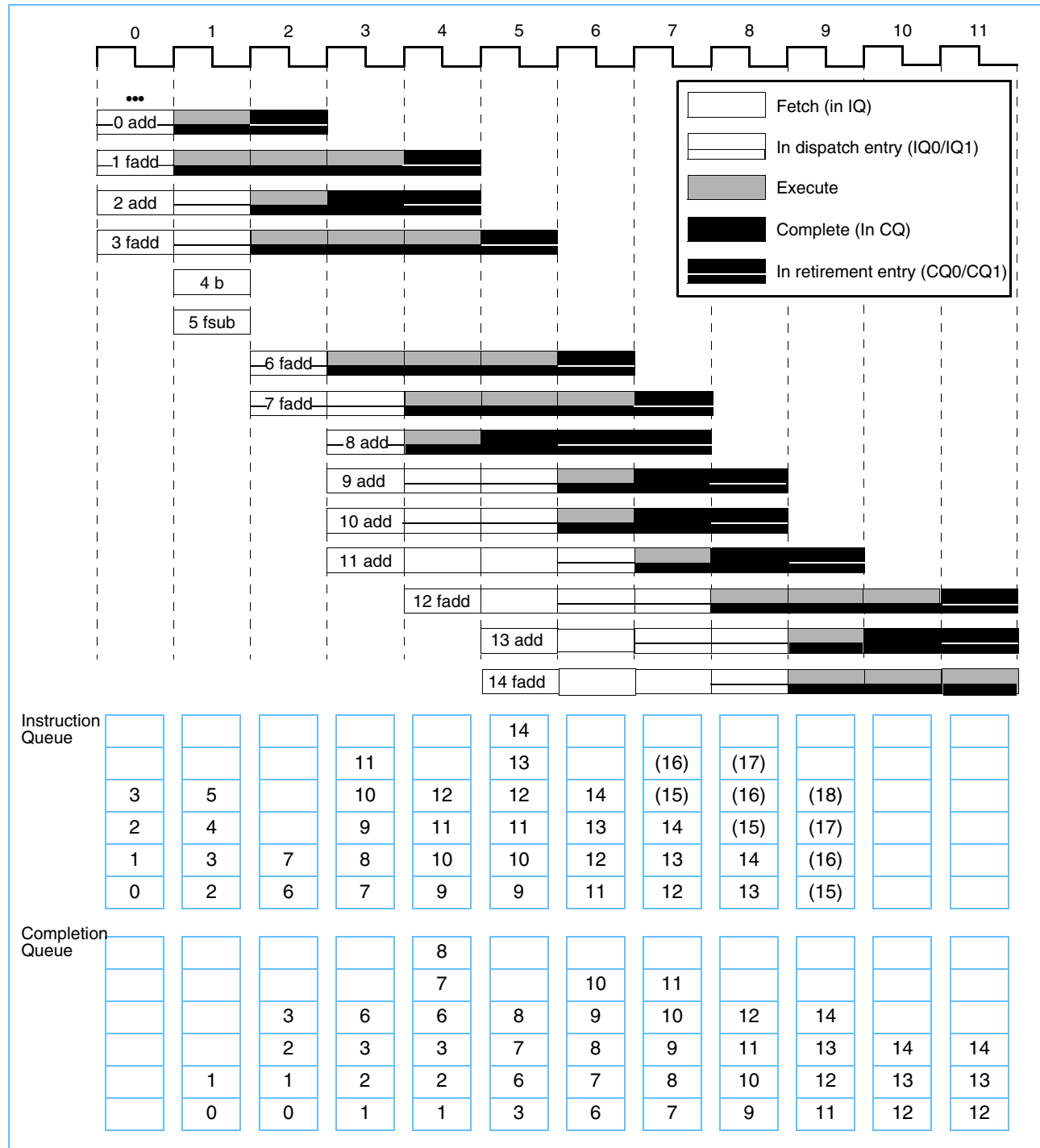


Figure 6-5 shows a simple example of instruction fetching that hits in the L1 cache. This example uses a series of integer add and double-precision floating-point add instructions to show how the number of instructions to be fetched is determined, how program order is maintained by the instruction and completion queues, how instructions are dispatched and retired in pairs (maximum), and how the FPU, IU1, and IU2 pipelines function. The following instruction sequence is examined:

```
0    add
1    fadd
2    add
3    fadd
4    br 6
5    fsub
6    fadd
7    fadd
8    add
9    add
10   add
11   add
12   fadd
13   add
14   fadd
15   .
16   .
17   .
```


Figure 6-5. Instruction Timing - Cache Hit



The instruction timing for this example is described cycle-by-cycle as follows:

0. In cycle 0, instructions 0 - 3 are fetched from the instruction cache. Instructions 0 and 1 are placed in two entries in the instruction queue from which they can be dispatched on the next clock cycle.
1. In cycle 1, instructions 0 and 1 are dispatched to the IU2 and FPU, respectively. Notice that for instructions to be dispatched they must be assigned positions in the completion queue. In this case, because the completion queue was empty, instructions 0 and 1 take the two lowest entries in the completion queue. Instructions 2 and 3 drop into the two dispatch positions in the instruction queue. Because there were two positions available in the instruction queue in clock cycle 0, two instructions (4 and 5) are fetched into the instruction queue. Instruction 4 is a branch unconditional instruction, which resolves immediately as taken. Because the branch is taken, it can therefore be folded from the instruction queue.
2. In cycle 2, assume a BTIC hit occurs and target instructions 6 and 7 are fetched into the instruction queue, replacing the folded instruction (4) and instruction 5. Instruction 0 completes, writes back its results, and vacates the completion queue by the end of the clock cycle. Instruction 1 enters the second FPU execute stage, instruction 2 is dispatched to the IU2, and instruction 3 is dispatched into the first FPU execute stage. Because the taken branch instruction (4) does not update either CTR or LR, it does not require a position in the completion queue and can be folded.
3. In cycle 3, target instructions (6 and 7) are fetched, replacing instructions 4 and 5 in IQ0 and IQ1. This replacement on taken branches is called branch folding. Instruction 1 proceeds through the last of the three FPU execute stages. Instruction 2 has executed but must remain in the completion queue until instruction 1 completes. Instruction 3 replaces instruction 1 in the second stage of the FPU, and instruction 6 replaces instruction 3 in the first stage.

Because there were four vacancies in the instruction queue in the previous clock cycle, instructions 8 - 1 are fetched in this clock cycle.

4. Instruction 1 completes in cycle 4, allowing instruction 2 to complete. Instructions 3 and 6 continue through the FPU pipeline. Because there were two openings in the completion queue in the previous cycle, instruction 7 is dispatched to the FPU and instruction 8 is dispatched to the IU2, filling the completion queue. Similarly, because there was one opening in the instruction queue in clock cycle 3, one instruction is fetched.
5. In cycle 5, instruction 3 completes, and instructions 13 and 14 are fetched. Instructions 6 and 7 continue through the FPU pipeline. No instructions are dispatched in this clock cycle because there were no vacant CQ entries in cycle 4.
6. In cycle 6, instruction 6 is completed and instruction 7 is in stage 3 of the FPU execute stage. Although instruction 8 has executed, it must wait for instruction 7 to complete. The two integer instructions, 9 and 10, are dispatched to the IU2 and IU1, respectively. No instructions are fetched because the instruction queue was full on the previous cycle.
7. In cycle 7, instruction 7 completes, allowing instruction 8 to complete as well. Instructions 9 and 10 remain in the completion stage, because at most two instructions can complete in a cycle. Because there was one opening in the completion queue in cycle 6, instruction 11 is dispatched to the IU2. Two more instructions (15 and 16, which are shown only in the instruction queue) are fetched.
8. In cycle 8, instructions 9 - 11 are through executing. Instructions 9 and 10 complete, write back, and vacate the completion queue. Instruction 11 must wait to complete on the following cycle. Because the completion queue had one opening in the previous cycle, instruction 12 can be dispatched to the FPU. Similarly, the instruction queue had one opening in the previous cycle, so one additional instruction, 17, can be fetched.

9. In cycle 9, instruction 11 completes, instruction 12 continues through the FPU pipeline, and instructions 13 and 14 are dispatched. One new instruction, 18, can be fetched on this cycle because the instruction queue had one opening on the previous clock cycle.

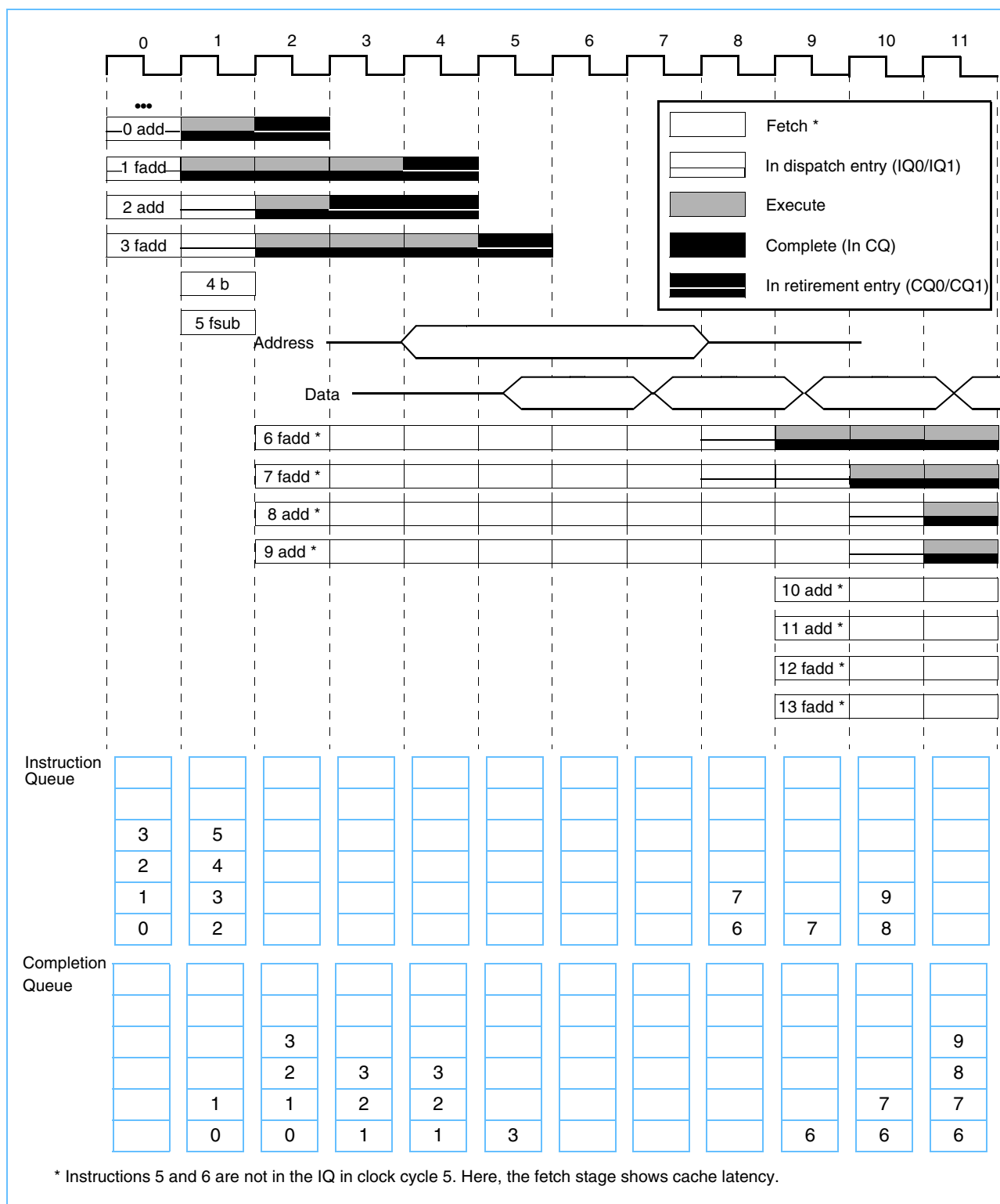
6.3.2.3 Cache Miss

Figure 6-6 on page 196 shows an instruction fetch that misses both the L1 cache and L2 cache. A processor/bus clock ratio of 1:2 is used. The same instruction sequence is used as in *Section 6.3.2.2* on page 190. However in this example, the branch target instruction is not in either the L1 or L2 cache.

A cache miss extends the latency of the fetch stage. In this example the fetch stage shown represents not only the time the instruction spends in the IQ, but the time required for the instruction to be loaded from system memory, beginning in clock cycle 2.

During clock cycle 3, the target instruction for the **b** instruction is not in the BTIC, the instruction cache or the L2 cache; therefore, a memory access must occur. During clock cycle 5, the address of the block of instructions is sent to the system bus. During clock cycle 7, two instructions (64 bits) are returned from memory on the first beat and are forwarded both to the cache and the instruction fetcher.

Figure 6-6. Instruction Timing - Cache Miss



6.3.2.4 L2 Cache Access Timing Considerations

If an instruction fetch misses both the BTIC and the L1 instruction cache, Espresso next looks in the L2 cache. If the requested instructions are there, they are burst into the Espresso core in much the same way as shown in *Figure 6-6* on page 196.

The latency for an instruction access that hits in the L2 cache is seven cycles longer than for a hit in the L1 cache.

6.3.2.5 Instruction Dispatch and Completion Considerations

Several factors affect the ability of Espresso to dispatch instructions at a peak rate of two per cycle: the availability of the execution unit, destination rename registers, and completion queue, as well as the handling of completion-serialized instructions. Several of these limiting factors are illustrated in the previous instruction timing examples.

To reduce dispatch unit stalls due to instruction data dependencies, Espresso provides a single-entry reservation station for the FPU, SRU, and each IU, and a 2-entry reservation station for the LSU. If a data dependency keeps an instruction from starting execution, that instruction is dispatched to the reservation station associated with its execution unit (and the rename registers are assigned), thereby freeing the positions in the instruction queue so instructions can be dispatched to other execution units. Execution begins during the same clock cycle that the rename buffer is updated with the data the instruction is dependent on.

If both instructions in IQ0 and IQ1 require the same execution unit they must be executed sequentially where IQ1 follows IQ0 through the execution unit. If these instructions require different execution units, they can be dispatched on the same cycle, and execute in parallel on separate execution units. They can complete together and be retired together on the same cycle.

The completion unit maintains program order after instructions are dispatched from the instruction queue, guaranteeing in-order completion and a precise exception model. Completing an instruction implies committing execution results to the architected destination registers. In-order completion ensures the correct architectural state when the Espresso core must recover from a mispredicted branch or an exception.

The instruction state and all information required for completion is kept in the 6-entry, first-in/first-out completion queue. A completion queue entry is allocated for each instruction when it is dispatched to an execution unit; if no entry is available, the dispatch unit stalls. A maximum of two instructions per cycle can be completed and retired from the completion queue, and the flow of instructions can stall when a longer-latency instruction reaches the last position in the completion queue. Subsequent instructions cannot be completed and retired until that longer-latency instruction completes and retires. Examples of this are shown in *Section 6.3.2.2* on page 190 and *Section 6.3.2.3* on page 195.

Espresso can execute instructions out-of-order, but in-order completion by the completion unit ensures a precise exception mechanism. Program-related exceptions are signaled when the instruction causing the exception reaches the last position in the completion queue. By this time, previous instructions are retired.

6.3.2.6 Rename Register Operation

To avoid contention for a given register file location in the course of out-of-order execution, the Espresso core provides rename registers for holding instruction results before the completion commits them to the architected register. There are six GPR rename registers, six FPR rename registers, and one each for the CR, LR, and CTR.

When the dispatch unit dispatches an instruction to its execution unit, it allocates a rename register (or registers) for the results of that instruction. If an instruction is dispatched to a reservation station associated with an execution unit due to a data dependency, the dispatcher also provides a tag to the execution unit identifying the rename register that forwards the required data at completion. When the source data reaches the rename register, execution can begin.

Instruction results are transferred from the rename registers to the architected registers by the completion unit when an instruction is retired from the completion queue, providing no exceptions proceed it and also any predicted branch conditions have been resolved correctly. If a branch prediction was incorrect, the instructions fetched along the predicted path are flushed from the completion queue, and any results of those instructions are flushed from the rename registers.

6.3.2.7 Instruction Serialization

Although the Espresso core can dispatch and complete two instructions per cycle, so-called serializing instructions limit dispatch and completion to one instruction per cycle. There are three types of instruction serialization:

- Execution serialization. Execution-serialized instructions are dispatched, held in the functional unit, and do not execute until all prior instructions have completed. A functional unit holding an execution-serialized instruction will not accept additional instructions from the dispatcher. For example, execution serialization is used for instructions that modify non-renamed resources. Results from these instructions are generally not available or forwarded to subsequent instructions until the instruction completes (using **mtspr** to write to LR or CTR does provide forwarding to branch instructions).
- Completion serialization (also referred to as post-dispatch or tail serialization). Completion-serialized instructions inhibit dispatching of subsequent instructions until the serialized instruction completes. Completion serialization is used for instructions that bypass the normal rename mechanism.
- Refetch serialization (flush serialization). Refetch-serialized instructions inhibit dispatch of subsequent instructions and force refetching of subsequent instructions after completion.

6.4 Execution Unit Timings

The following sections describe instruction timing considerations within each of the respective execution units in the Espresso core.

6.4.1 Branch Processing Unit Execution Timing

Flow control operations (conditional branches, unconditional branches, and traps) are typically expensive to execute in most machines because they disrupt normal flow in the instruction stream. When a change in program flow occurs, the IQ must be reloaded with the target instruction stream. Previously issued instructions continue to execute while the new instruction stream makes its way into the IQ. However, depending on whether the target instruction is in the BTIC, instruction L1 cache, L2 cache, or in system memory, some opportunities might be missed to execute instructions, as the example in *Section 6.3.2.3 Cache Miss* on page 195 shows.

Performance features such as the branch folding, BTIC, dynamic branch prediction (implemented in the BHT), 2-level branch prediction, and the implementation of nonblocking caches minimize the penalties associated with flow control operations on Espresso. The timing for branch instruction execution is determined by many factors including the following:

- Whether the branch is taken
- Whether instructions in the target stream, typically the first two instructions in the target stream, are in the branch target instruction cache (BTIC)
- Whether the target instruction stream is in the L1 cache
- Whether the branch is predicted
- Whether the prediction is correct

6.4.1.1 Branch Folding

When a branch instruction is encountered by the fetcher, the BPU immediately begins to decode it and tries to resolve it. Branch folding is the removal of branches from the instruction stream. This is independent of whether the branch is taken or not taken. However, if the branch instruction updates either the LR or CTR, it cannot be removed and must be allocated a position in the completion queue. If a branch cannot be resolved, immediately, it is predicted. Instruction fetching resumes along the predicted path, and those instructions are conditionally fed into the instruction queue. Later, if the prediction is finally resolved correctly, the fetched instructions are validated and allowed to complete and be retired. If the prediction is resolved incorrectly, instructions fetched are invalidated and instruction fetching resumes along the other path of the branch.

Figure 6-7 on page 200 shows branch folding. Here a **br** instruction is encountered in a series of **add** instructions. The branch is resolved as taken. What happens on the next clock cycle depends on whether the target instruction stream is in the BTIC, the instruction L1 cache, or if it must be fetched from the L2 cache or from system memory. *Figure 6-7* shows cases where there is a BTIC hit, and when there is a BTIC miss (and instruction cache hit).

If there is a BTIC hit on the next clock cycle, the **b** instruction is replaced by the target instruction, **and1**, that was found in the BTIC; the second **and** instruction is also fetched from the BTIC. On the next clock cycle, the next four **and** instructions from the target stream are fetched from the instruction cache.

If the target instruction is not in the BTIC, there is an idle cycle while the fetcher attempts to fetch the first four instructions from the instruction cache (on the next clock cycle). In the example in *Figure 6-7*, the first four target instructions are fetched on the next clock.

If it misses in the BTIC or L1 caches, an L2 cache or memory access is required, the latency of which is dependent on several factors, such as processor/bus clock ratios. In most cases, new instructions arrive in the IQ before the execution units become idle.

Figure 6-7. Branch Taken

Branch Folding (Taken Branch/BTIC Hit)				Branch Folding (Taken Branch/BTIC Miss)			
	Clock 0	Clock 1	Clock 2		Clock 0	Clock 1	Clock 2
IQ5	add5			IQ5	add5		
IQ4	add4			IQ4	add4		
IQ3	add3		and6	IQ3	add3		and4
IQ2	b		and5	IQ2	b		and3
IQ1	add2	and2	and4	IQ1	add2		and2
IQ0	add1	and1	and3	IQ0	add1		and1

Figure 6-8 shows the removal of fall-through branch instructions, which occurs when a branch is not taken or is predicted as not taken.

Figure 6-8. Removal of Fall-Through Branch Instruction

Branch Fall-Through (Not-Taken Branch)			
	Clock 0	Clock 1	Clock 2
IQ5	add5		
IQ4	add4		
IQ3	add3	add5	add7
IQ2	b	add4	add6
IQ1	add2	add3	add5
IQ0	add1	b	add4

When a branch instruction is detected before it reaches a dispatch position, and if the branch is correctly predicted as taken, folding the branch instruction (and any instructions from the incorrect path) reduces the latency required for flow control to zero; instruction execution proceeds as though the branch was never there.

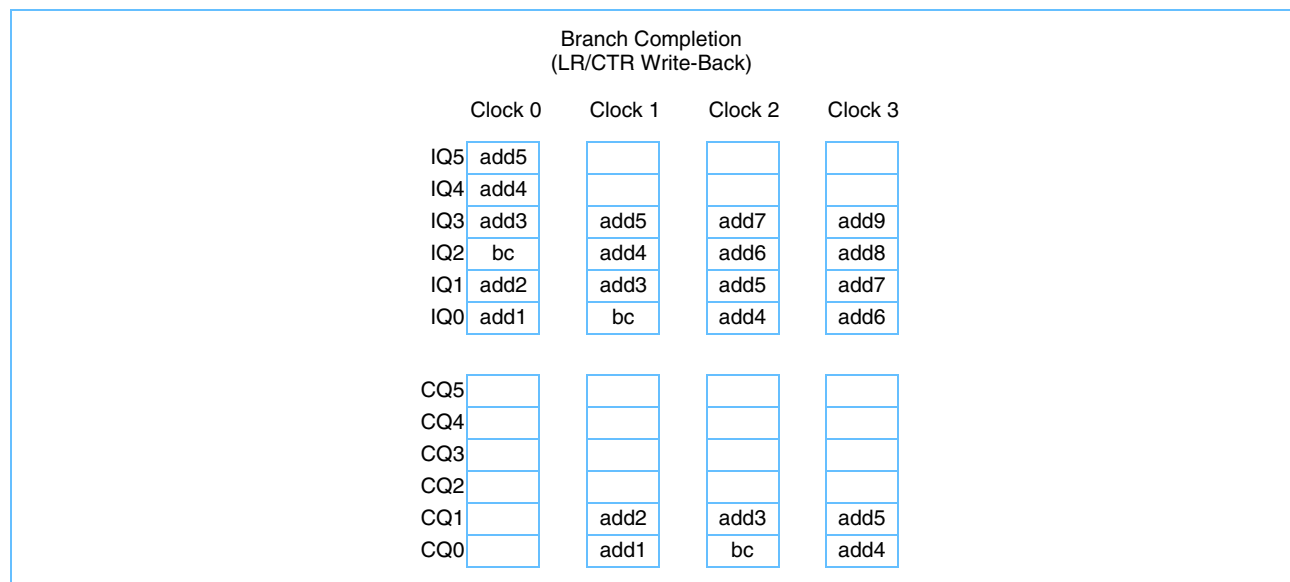
The advantage of removing the fall-through branch instructions at dispatch is only marginally less than that of branch folding. Because the branch is not taken, only the branch instruction needs to be discarded. The only cost of expelling the branch instruction from one of the dispatch entries rather than folding it is missing a chance to dispatch an executable instruction from that position.

6.4.1.2 Branch Instructions and Completion

As described in the previous section, instructions that do not update either the LR or CTR are removed from the instruction stream before they reach the completion queue, either for branch taken or by removing fall-through branch instructions at dispatch. However, branch instructions that update the architected LR and CTR must do so in program order and therefore must perform write-back in the completion stage, like the instructions that update the FPRs and GPRs.

Branch instructions that update the CTR or LR pass through the instruction queue like nonbranch instructions. At the point of dispatch, however, they are not sent to an execution unit, but rather are assigned a slot in the completion queue, as shown in *Figure 6-9*.

Figure 6-9. Branch Completion



In this example, the **bc** instruction is encoded to decrement the CTR. It is predicted as not-taken in clock cycle 0. In clock cycle 2, **bc** and **add3** are both dispatched. In clock cycle 3, the architected CTR is updated and the **bc** instruction is retired from the completion queue.

6.4.1.3 Branch Prediction and Resolution

The Espresso core supports the following two types of branch prediction:

- Static branch prediction: This is defined by the PowerPC Architecture as part of the encoding of branch instructions.
- Dynamic branch prediction: This is a processor-specific mechanism implemented in hardware (in particular the branch history table, or BHT) that monitors branch instruction behavior and maintains a record from which the next occurrence of the branch instruction is predicted.

When a conditional branch cannot be resolved due to a CR data dependency, the BPU predicts whether it will be taken, and instruction fetching proceeds down the predicted path. If the branch prediction resolves as incorrect, the instruction queue and all subsequently executed instructions are purged, instructions executed before the predicted branch are allowed to complete, and instruction fetching resumes down the correct path.

Espresso executes through two levels of prediction. Instructions from the first unresolved branch can execute, but they cannot be retired until the branch is resolved. If a second branch instruction is encountered in the predicted instruction stream, it can be predicted and instructions can be fetched, but not executed, from the second branch. No action can be taken for a third branch instruction until at least one of the two previous branch instructions is resolved.

The number of instructions that can be executed after the issue of a predicted branch instruction is limited by the fact that no instruction executed after a predicted branch can actually update (be retired) the register files or memory until the branch is resolved. That is, instructions can be issued and executed, but cannot be retired from the completion unit. When an instruction following a predicted branch completes execution, it does not write back its results to the architected registers; instead, it stalls in the completion queue. Of course, when the completion queue is full, no additional instructions can be dispatched, even if an execution unit is idle.

In the case of a misprediction, Espresso can easily redirect the instruction stream because the programming model has not been updated. When a branch is mispredicted, all instructions that were dispatched after the predicted branch instruction are flushed from the completion queue and any results are flushed from the rename registers.

The BTIC is a cache of two recently used instructions at the target (branch to address) of branch instructions. If a taken-branch hits in the BTIC, two instructions are fed into the instruction queue on the next cycle. If a taken-branch misses in the BTIC instruction fetching is done from the L1 instruction cache. Coherency of the BTIC table is maintained by table reset on an I-Cache flush invalidate, **icbi** or **rfi** instruction execution, or when an exception is taken.

In some situations, an instruction sequence creates dependencies that keep a branch instruction from being predicted because the address for the target of the branch is not available. This delays execution of the subsequent instruction stream. The instruction sequences and the resulting action of the branch instruction are described as follows:

- An **mtspr** (LK) followed by a **bclr**. Fetching stops and the branch waits for the **mtspr** to execute.
- An **mtspr** (CTR) followed by a **bcctr**. Fetching stops and the branch waits for the **mtspr** to execute.
- An **mtspr** (CTR) followed by a **bc** (CTR decrement). Fetching stops and the branch waits for the **mtspr** to execute.
- A third **bc** (based-on-CR) is encountered while there are two unresolved **bc** (based-on-CR). The third **bc** (based-on-CR) is not executed and fetching stops until one of the previous **bc** (based-on-CR) is resolved. (Note that branch conditions can be a function of the CTR and the CR; if the CTR condition is sufficient to resolve the branch, then a CR-dependency is ignored.)

Static Branch Prediction

The PowerPC Architecture provides a field in branch instructions (the BO field) to allow software to speculate (hint) whether a branch is likely to be taken. Rather than delaying instruction processing until the condition is known, Espresso uses the instruction encoding to predict whether the branch is likely to be taken and begins fetching and executing along that path. When the branch condition is known, the prediction is evaluated. If the prediction was correct, program flow continues along that path; otherwise, the processor flushes any instructions and their results from the mispredicted path, and program flow resumes along the correct path.

Static branch prediction is used when **HID0[BHT]** is cleared. That is, the branch history table, which is used for dynamic branch prediction, is disabled.

For information about static branch prediction, see the conditional branch control section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Predicted Branch Timing Examples

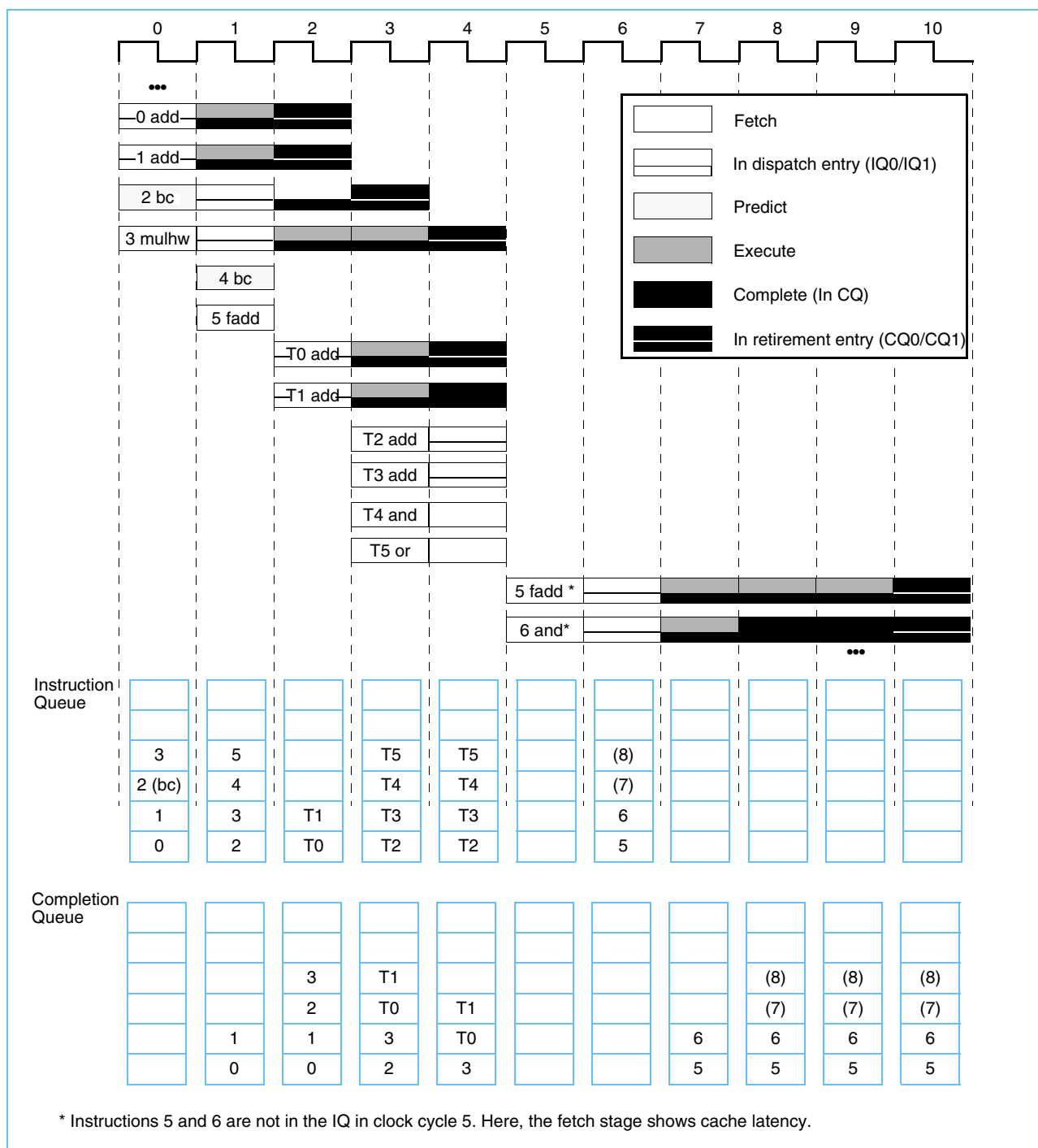
Figure 6-10 on page 204 shows cases where branch instructions are predicted. It shows how both taken and not-taken branches are handled and how Espresso handles both correct and incorrect predictions. The example shows the timing for the following instruction sequence:

```

0    add
1    add
2    bc
3    mulhw
4    bc T0
5    fadd
6    and
   add
T7   add
T8   add
T9   add
T10  add
T11  or

```

Figure 6-10. Branch Instruction Timing



0. During clock cycle 0, instructions 0 and 1 are dispatched to their respective execution units. Instruction 2 is a branch instruction that updates the CTR. It is predicted as not taken in clock cycle 0. Instruction 3 is a **mulhw** instruction on which instruction 4 depends.
1. In clock cycle 1, instructions 2 and 3 enter the dispatch entries in the IQ. Instruction 4 (a second **bc** instruction) and 5 are fetched. The second **bc** instruction is predicted as taken. It can be folded, but it cannot be resolved until instruction 3 writes back.
2. In clock cycle 2, instruction 4 has been folded and instruction 5 has been flushed from the IQ. The two target instructions, T0 and T1, are both in the BTIC, so that they are fetched in this cycle. Note that even though the first **bc** instruction might not have been resolved by this point (we can assume it has), Espresso allows fetching from a second predicted branch stream. However, these instructions can not be dispatched until the previous branch has resolved.
3. In clock cycle 3, target instructions T2 - T5 are fetched as T0 and T1 are dispatched.
4. In clock cycle 4, instruction 3, on which the second branch instruction depended, writes back and the branch prediction is proven incorrect. Even though T0 is in CQ1, from which it can be written back, it is not written back because the branch prediction was incorrect. All target instructions are flushed from their positions in the pipeline at the end of this clock cycle, as are any results in the rename registers.

After one clock cycle required to refetch the original instruction stream, instruction 5, the same instruction that was fetched in clock cycle 1, is brought back into the IQ from the instruction cache. Three other instructions, not all of which are shown, are also brought into the IQ from the instruction cache.

6.4.2 Integer Unit Execution Timing

The Espresso core has two integer units. The IU1 can execute all integer instructions, and the IU2 can execute all integer instructions except multiply and divide instructions. As shown in *Figure 6-2* on page 186, each integer unit has one execute pipeline stage; thus, when a multicycle (for example, divide) integer instruction is being executed, no additional integer instruction can begin to execute in that unit. However, the other unit IU2 can continue to execute integer instructions. *Table 6-6* on page 213 lists integer instruction latencies.

Most integer instructions have an execution latency of one clock cycle.

6.4.3 Floating-Point Unit Execution Timing

The floating-point unit on Espresso executes all floating-point instructions. Execution of most floating-point instructions is pipelined within the FPU, allowing up to three instructions to be executing in the FPU concurrently. While most floating-point instructions execute with 3- or 4-cycle latency, and 1- or 2-cycle throughput, two instructions (**fdivs** and **fdiv**) execute with latencies of 11 - 33 cycles. The **fdivs**, **fdiv**, **mtfsb0**, **mtfsb1**, **mtfsfi**, **mffs**, and **mtfsf** instructions block the floating-point unit pipeline until they complete execution, and thereby inhibit the dispatch of additional floating-point instructions. See *Figure 6-7* on page 214 for floating-point instruction execution timing.

6.4.4 Effect of Floating-Point Exceptions on Performance

For the fastest and most predictable floating-point performance, all exceptions should be disabled in the FPSCR and MSR.

6.4.5 Load/Store Unit Execution Timing

The execution of most load and store instructions is pipelined. The LSU has two pipeline stages. The first is for effective address calculation and MMU translation and the second is for accessing data in the cache. Load and store instructions have a 2-cycle latency and 1-cycle throughput. For instructions that store FPR values (**stfd**, **stfs**, and their variations), the data to be stored is prefetched from the source register during the first pipeline stage. In cases where this register is updated that same cycle, the instruction stalls to get the correct data, resulting in one additional cycle of latency.

If operands are misaligned, additional latency might be required either for an alignment exception to be taken or for additional bus accesses. Load instructions that miss in the cache, block subsequent cache accesses during the cache line refill. *Table 6-8* on page 216 gives load and store instruction execution latencies.

6.4.6 Effect of Operand Placement on Performance

The PowerPC VEA states that the placement (location and alignment) of operands in memory might affect the relative performance of memory accesses, and in some cases affect it significantly. The effects memory operand placement has on performance are shown in *Table 6-1*.

The best performance is guaranteed if memory operands are aligned on natural boundaries. For the best performance across the widest range of implementations, the programmer should assume the performance model described in the operand conventions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

The effect of misalignment on memory access latency is the same for big-endian and little-endian addressing modes except for multiple and string operations that cause an alignment exception in little-endian mode.

Table 6-1. Performance Effects of Memory Operand Placement (Page 1 of 2)

Operand		Boundary Crossing			
Size	Byte Alignment	None	8 Byte	Cache Block	Protection Boundary
Integer					
4 byte	4	Optimal ¹	—	—	—
	< 4	Optimal	Good	Good	Good
2 byte	2	Optimal	—	—	—
	< 2	Optimal	Good	Good	Good
1 byte	1	Optimal	—	—	—
lmw , stmw ²	4	Good ³	Good	Good	Good
	< 4	Poor ⁴	Poor	Poor	Poor
String ²	—	Good	Good	Good	Good

Note:

1. Optimal means one EA calculation occurs.
2. Not supported in little-endian mode; causes an alignment exception.
3. Good means multiple EA calculations occur that might cause additional bus activities with multiple bus transfers.
4. Poor means that an alignment exception occurs.

Table 6-1. Performance Effects of Memory Operand Placement (Page 2 of 2)

Operand		Boundary Crossing			
Size	Byte Alignment	None	8 Byte	Cache Block	Protection Boundary
Floating-Point					
8 byte	8	Optimal	—	—	—
	4	—	Good	Good	Good
	< 4	—	Poor	Poor	Poor
4 byte	4	Optimal	—	—	—
	< 4	Poor	Poor	Poor	Poor

Note:

1. Optimal means one EA calculation occurs.
2. Not supported in little-endian mode; causes an alignment exception.
3. Good means multiple EA calculations occur that might cause additional bus activities with multiple bus transfers.
4. Poor means that an alignment exception occurs.

6.4.7 Integer Store Gathering

The Espresso core performs store gathering for write-through operations to nonguarded space. It performs cache-inhibited stores to nonguarded space for 4-byte, word-aligned stores. These stores are combined in the LSU to form a doubleword and are sent out on the 60Xe bus as a single-beat operation. However, stores are gathered only if the successive stores meet the criteria and are queued and pending. Store gathering occurs regardless of the address order of the stores. Store gathering is enabled by setting HID0[SGE]. Stores can be gathered in both endian modes.

Store gathering is not done for the following:

- Cacheable store operations
- Stores to guarded cache-inhibited or write-through space
- Byte-reverse store operations
- **stwcx.** instructions
- **ecowx** instructions
- A store that occurs during a table search operation
- Floating-point store operations

If store gathering is enabled and the stores do not fall under the categories listed previously, an **eieio** or **sync** instruction must be used to prevent two stores from being gathered.

6.4.8 System Register Unit Execution Timing

Most instructions executed by the SRU either directly access renamed registers or access or modify nonrenamed registers. They generally execute in a serial manner. Results from these instructions are not available to subsequent instructions until the instruction completes and is retired. See *Section 6.3.2.7* for more information about serializing instructions executed by the SRU, and see *Table 6-4* and *Table 6-5* for SRU instruction execution timings.

6.5 Memory Performance Considerations

Because the Espresso core can have a maximum instruction throughput of three instructions per clock cycle, lack of memory bandwidth can affect performance. For Espresso to maximize performance, it must be able to read and write data efficiently. If a system has multiple bus devices, one of them might experience long memory latencies while another bus master (for example, a direct-memory access controller) is using the external bus.

6.5.1 Caching and Memory Coherency

To minimize the effect of bus contention, the PowerPC Architecture defines WIM bits that are used to configure memory regions as caching-enforced or caching-inhibited. Accesses to such memory locations never update the L1 cache. If a cache-inhibited access hits the L1 cache, the cache block is invalidated. If the cache block is marked modified, it is copied back to memory before being invalidated. Where caching is permitted, memory is configured as either write-back or write-through, which are described as follows:

- **Write-back:** Configuring a memory region as write-back lets a processor modify data in the cache without updating system memory. For such locations, memory updates occur only on modified cache block replacements, cache flushes, or when one processor needs data that is modified in another's cache. Therefore, configuring memory as write-back can help when bus traffic might cause bottlenecks, especially for multiprocessor systems and for regions in which data, such as local variables, is used often and is coupled closely to a processor.

If multiple devices use data in a memory region marked write-through, snooping must be enabled to allow the copy-back and cache invalidation operations necessary to ensure cache coherency. The Espresso snooping hardware keeps other devices from accessing invalid data. For example, when snooping is enabled, Espresso monitors transactions of other bus devices. For example, if another device needs data that is modified on the Espresso cache, the access is delayed so that Espresso can copy the modified data to memory.

- **Write-through:** Store operations to memory marked write-through always update both system memory and the L1 cache on cache hits. Because valid cache contents always match system memory marked write-through, cache hits from other devices do not cause modified data to be copied back as they do for locations marked write-back. However, all write operations are passed to the bus, which can limit performance. Load operations that miss the L1 cache must wait for the external store operation.

Write-through configuration is useful when cached data must agree with external memory (for example, video memory), when shared (global) data might be needed often, or when it is undesirable to allocate a cache block on a cache miss.

Figure 3-1 Cache Integration on page 112 describes the caches, memory configuration, and snooping in detail.

6.5.2 Effect of TLB Miss

If a page address translation is not in a TLB, the Espresso hardware searches the page tables and updates the TLB when a translation is found. *Table 6-2* shows the estimated latency for the hardware TLB load for different cache configurations and conditions.

Table 6-2. TLB Miss Latencies

L1 Condition (Instruction and Data)	L2 Condition	Processor/System Bus Clock Ratio	Estimated Latency (Cycles)
100% cache hit	—	—	7
100% cache miss	100% cache hit	—	15
100% cache miss	100% cache miss	2.5:1 (6:3:3:3 memory)	72
100% cache miss	100% cache miss	4:1 (5:2:2:2 memory)	87

The PTE table search assumes a hit in the first entry of the primary PTEG.

6.6 Instruction Scheduling Guidelines

The performance of the Espresso core can be improved by avoiding resource conflicts and scheduling instructions to take fullest advantage of the parallel execution units. Instruction scheduling on Espresso can be improved by observing the following guidelines:

- To reduce mispredictions, separate the instruction that sets CR bits from the branch instruction that evaluates them. Because there can be no more than 12 instructions in the processor (with the instruction that sets CR in CQ0 and the dependent branch instruction in IQ5), there is no advantage to having more than 10 instructions between them.
 - Likewise, when branching to a location specified by the CTR or LR, separate the **mtspr** instruction that initializes the CTR or LR from the dependent branch instruction. This ensures the register values are available sooner to the branch instruction.
 - Schedule instructions such that two can be dispatched at a time.
 - Schedule instructions to minimize stalls due to execution units being busy.
 - Avoid scheduling high-latency instructions close together. Interspersing single-cycle latency integer instructions between longer-latency instructions minimizes the effect that instructions such as integer divide and multiply can have on throughput.
 - Avoid using serializing instructions.
 - Schedule instructions to avoid dispatch stalls:
 - Six instructions can be tracked in the completion queue; therefore, only six instructions can be in the execute stages at any one time.
 - There are six GPR rename registers; therefore only six GPRs can be specified as destination operands at any time. If no rename registers are available, instructions cannot enter the execute stage and remain in the reservation station or instruction queue until they become available.
- Note:** Load with update address instructions use two rename registers.
- Similarly, there are six FPR rename registers, so that only six FPR destination operands can be in the execute and complete stages at any time.

6.6.1 Branch, Dispatch, and Completion Unit Resource Requirements

This section describes the specific resources required to avoid stalls during branch resolution, instruction dispatching, and instruction completion.

6.6.1.1 Branch Resolution Resource Requirements

The following is a list of branch instructions and the resources required to avoid stalling the fetch unit in the course of branch resolution:

- The **bclr** instruction requires LR availability.
- The **bcctr** instruction requires CTR availability.
- Branch and link instructions require shadow LR availability.
- The “branch conditional on counter decrement and the CR” condition requires CTR availability or the CR condition must be false, and Espresso cannot execute instructions after an unresolved predicted branch when the BPU encounters a branch.
- A branch conditional on CR condition cannot be executed following an unresolved predicted branch instruction.

6.6.1.2 Dispatch Unit Resource Requirements

The following is a list of resources required to avoid stalls in the dispatch unit. IQ0 and IQ1 are the two dispatch entries in the instruction queue:

- Requirements for dispatching from IQ0 are as follows:
 - Needed execution unit is available.
 - Needed GPR rename registers are available.
 - Needed FPR rename registers are available.
 - Completion queue is not full.
 - A completion-serialized instruction is not being executed.
- Requirements for dispatching from IQ1 are as follows:
 - Instruction in IQ0 must dispatch.
 - Instruction dispatched by IQ0 is not completion- or refetch-serialized.
 - Needed execution unit is available (after dispatch from IQ0).
 - Needed GPR rename registers are available (after dispatch from IQ0).
 - Needed FPR rename register is available (after dispatch from IQ0).
 - Completion queue is not full (after dispatch from IQ0).

6.6.1.3 Completion Unit Resource Requirements

The following is a list of resources required to avoid stalls in the completion unit; note that the two completion entries are described as CQ0 and CQ1, where CQ0 is the completion queue located at the end of the completion queue (see *Figure 6-4* on page 191).

- Requirements for completing an instruction from CQ0 are as follows:
 - Instruction in CQ0 must be finished.
 - Instruction in CQ0 must not follow an unresolved predicted branch.
 - Instruction in CQ0 must not cause an exception.

- Requirements for completing an instruction from CQ1 are as follows:
 - Instruction in CQ0 must complete in same cycle.
 - Instruction in CQ1 must be finished.
 - Instruction in CQ1 must not follow an unresolved predicted branch.
 - Instruction in CQ1 must not cause an exception.
 - Instruction in CQ1 must be an integer or load instruction.
 - Number of CR updates from both CQ0 and CQ1 must not exceed two.
 - Number of GPR updates from both CQ0 and CQ1 must not exceed two.
 - Number of FPR updates from both CQ0 and CQ1 must not exceed two.

6.7 Instruction Latency Summary

Table 6-3 through Table 6-8 on page 216 list the latencies associated with instructions executed by each execution unit. Table 6-3 describes branch instruction latencies.

Table 6-3. Branch Instructions

Mnemonic	Primary	Extended	Latency
b[l][a]	18	—	Unless these instructions update either the CTR or the LR, branch operations are folded if they are either taken or predicted as taken. They fall through if they are not taken or predicted as not taken.
bc[l][a]	16	—	
bcctr[l]	19	528	
bclr[l]	19	16	

Table 6-4 lists system register instruction latencies.

Table 6-4. System Register Instructions (Page 1 of 2)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
eieio	31	854	SRU	1	—
isync	19	150	SRU	2	Completion, refetch
mfmsr	31	83	SRU	1	—
mfspr (iBATs)	31	339	SRU	3	—
mfspr (DBATs, GQRs)	31	339	SRU	3	Execution
mfspr (shared) ¹	31	339	SRU	3	Execution
mfspr (THRMs, DABR, SDR1)	31	339	SRU	2	Execution
mfspir (others)	31	339	SRU	1	Execution
mfsr	31	595	SRU	3	—
mfsrin	31	659	SRU	3	Execution
mftb	31	371	SRU	3	—
mtmsr	31	146	SRU	2	Execution

Notes:

1. Shared registers are those that are each implemented as a single register that is shared by all cores.
2. This assumes no pending stores in the store queue. If there are pending stores, the **sync** completes after they complete to memory. If broadcast is enabled on the 60Xe bus, **sync** completes only after a successful broadcast.
3. **tlbsync** is dispatched only to the completion buffer (not to any execution unit) and is marked finished as it is dispatched. Upon retirement, it waits for any pending **tlbie** operations in other cores to finish before completing.

Table 6-4. System Register Instructions (Page 2 of 2)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
mtspr (BATs)	31	467	SRU	2	Execution
mtspr (shared) ¹	31	467	SRU	2	Execution
mtspr (others)	31	467	SRU	1	Execution
mtsr	31	210	SRU	2	Execution
mtsrin	31	242	SRU	2	Execution
mttb	31	467	SRU	2	Execution
rfi	19	50	SRU	2	Completion, refetch
sc	17	- -1	SRU	2	Completion, refetch
sync	31	598	SRU	3 ²	—
tlbsync ³	31	566	—	—	

Notes:

1. Shared registers are those that are each implemented as a single register that is shared by all cores.
2. This assumes no pending stores in the store queue. If there are pending stores, the **sync** completes after they complete to memory. If broadcast is enabled on the 60Xe bus, **sync** completes only after a successful broadcast.
3. **tlbsync** is dispatched only to the completion buffer (not to any execution unit) and is marked finished as it is dispatched. Upon retirement, it waits for any pending **tlbie** operations in other cores to finish before completing.

Table 6-5 lists Condition Register logical instruction latencies.

Table 6-5. Condition Register Logical Instructions

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
crand	19	257	SRU	1	Execution
crandc	19	129	SRU	1	Execution
creqv	19	289	SRU	1	Execution
crnand	19	225	SRU	1	Execution
crnor	19	33	SRU	1	Execution
cror	19	449	SRU	1	Execution
crorc	19	417	SRU	1	Execution
crxor	19	193	SRU	1	Execution
mcrf	19	0	SRU	1	Execution
mcrxr	31	512	SRU	1	Execution
mfcr	31	19	SRU	1	Execution
mtcrf	31	144	SRU	1	Execution

Table 6-6 on page 213 shows integer instruction latencies. Note that the IU1 executes all integer arithmetic instructions: multiply, divide, shift, rotate, add, subtract, and compare. The IU2 executes all integer instructions except multiply and divide (that is, shift, rotate, add, subtract, and compare).

Table 6-6. Integer Instructions (Page 1 of 2)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
addc[o][.]	31	10	IU1/IU2	1	—
adde[o][.]	31	138	IU1/IU2	1	Execution
addi	14	—	IU1/IU2	1	—
addic	12	—	IU1/IU2	1	—
addic.	13	—	IU1/IU2	1	—
addis	15	—	IU1/IU2	1	—
addme[o][.]	31	234	IU1/IU2	1	Execution
addze[o][.]	31	202	IU1/IU2	1	Execution
add[o][.]	31	266	IU1/IU2	1	—
andc[.]	31	60	IU1/IU2	1	—
andi.	28	—	IU1/IU2	1	—
andis.	29	—	IU1/IU2	1	—
and[.]	31	28	IU1/IU2	1	—
cmp	31	0	IU1/IU2	1	—
cmpi	11	—	IU1/IU2	1	—
cmpl	31	32	IU1/IU2	1	—
cmpli	10	—	IU1/IU2	1	—
cntlzw[.]	31	26	IU1/IU2	1	—
divwu[o][.]	31	459	IU1	19	—
divw[o][.]	31	491	IU1	19	—
eqv[.]	31	284	IU1/IU2	1	—
extsb[.]	31	954	IU1/IU2	1	—
extsh[.]	31	922	IU1/IU2	1	—
mulhwu[.]	31	11	IU1	2, 3, 4, 5, 6	—
mulhw[.]	31	75	IU1	2, 3, 4, 5	—
mulli	7	—	IU1	2, 3	—
mull[o][.]	31	235	IU1	2, 3, 4, 5	—
nand[.]	31	476	IU1/IU2	1	—
neg[o][.]	31	104	IU1/IU2	1	—
nor[.]	31	124	IU1/IU2	1	—
orc[.]	31	412	IU1/IU2	1	—
ori	24	—	IU1/IU2	1	—
oris	25	—	IU1/IU2	1	—
or[.]	31	444	IU1/IU2	1	—
rlwimi[.]	20	—	IU1/IU2	1	—
rlwinm[.]	21	—	IU1/IU2	1	—

Table 6-6. Integer Instructions (Page 2 of 2)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
rlwnm [.]	23	—	IU1/IU2	1	—
slw [.]	31	24	IU1/IU2	1	—
srawi [.]	31	824	IU1/IU2	1	—
sraw [.]	31	792	IU1/IU2	1	—
srw [.]	31	536	IU1/IU2	1	—
subfc [o][.]	31	8	IU1/IU2	1	—
subfe [o][.]	31	136	IU1/IU2	1	Execution
subfc	8	—	IU1/IU2	1	—
subfme [o][.]	31	232	IU1/IU2	1	Execution
subfze [o][.]	31	200	IU1/IU2	1	Execution
subf [.]	31	40	IU1/IU2	1	—
tw	31	4	IU1/IU2	2	—
twi	3	—	IU1/IU2	2	—
xori	26	—	IU1/IU2	1	—
xoris	27	—	IU1/IU2	1	—
xor [.]	31	316	IU1/IU2	1	—

Table 6-7 shows latencies for floating-point instructions. Pipelined floating-point instructions are shown with the number of clocks in each pipeline stage separated by dashes. Floating-point instructions with a single entry in the cycles column are not pipelined; when the FPU executes these nonpipelined instructions, it remains busy for the full duration of the instruction execution and is not available for subsequent instructions.

Table 6-7. Floating-Point Instructions (Page 1 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
fabs [.]	63	264	FPU	1-1-1	—
fadds [.]	59	21	FPU	1-1-1	—
fadd [.]	63	21	FPU	1-1-1	—
fcmpo	63	32	FPU	1-1-1	—
fcmpu	63	0	FPU	1-1-1	—
fctiwz [.]	63	15	FPU	1-1-1	—
fctiw [.]	63	14	FPU	1-1-1	—
fdivs [.]	59	18	FPU	17	—
fdiv [.]	63	18	FPU	31	—
fmadds [.]	59	29	FPU	1-1-1	—
fmadd [.]	63	29	FPU	2-1-1	—
fmr [.]	63	72	FPU	1-1-1	—
fmsubs [.]	59	28	FPU	1-1-1	—
fmsub [.]	63	28	FPU	2-1-1	—

Table 6-7. Floating-Point Instructions (Page 2 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
fmuls[.]	59	25	FPU	1-1-1	—
fmul[.]	63	25	FPU	2-1-1	—
fnabs[.]	63	136	FPU	1-1-1	—
fneg[.]	63	40	FPU	1-1-1	—
fnmadds[.]	59	31	FPU	1-1-1	—
fnmadd[.]	63	31	FPU	2-1-1	—
fnmsubs[.]	59	30	FPU	1-1-1	—
fnmsub[.]	63	30	FPU	2-1-1	—
fres[.]	59	24	FPU	2-1-1	—
frsp[.]	63	12	FPU	1-1-1	—
frsqrte[.]	63	26	FPU	2-1-1	—
fsel[.]	63	23	FPU	1-1-1	—
fsubs[.]	59	20	FPU	1-1-1	—
ps_abs[.]	4	264	FPU	1-1-1	—
ps_add[.]	4	21	FPU	1-1-1	—
ps_cmpo0	4	32	FPU	1-1-1	—
ps_cmpo1	4	96	FPU	1-1-1	—
ps_cmpu0	0	0	FPU	1-1-1	—
ps_cmpu1	4	64	FPU	1-1-1	—
ps_div[.]	4	18	FPU	17	—
ps_madd[.]	4	29	FPU	1-1-1	—
ps_madds0[.]	4	14	FPU	1-1-1	—
ps_madds1[.]	4	15	FPU	1-1-1	—
ps_merge00[.]	4	528	FPU	1-1-1	—
ps_merge01[.]	4	560	FPU	1-1-1	—
ps_merge10[.]	4	592	FPU	1-1-1	—
ps_merge_11[.]	4	624	FPU	1-1-1	—
ps_mr[.]	4	72	FPU	1-1-1	—
ps_msub[.]	4	28	FPU	1-1-1	—
ps_mul[.]	4	25	FPU	1-1-1	—
ps_muls0[.]	4	12	FPU	1-1-1	—
ps_muls1[.]	4	13	FPU	1-1-1	—
ps_nabs[.]	4	136	FPU	1-1-1	—
ps_neg[.]	4	40	FPU	1-1-1	—
ps_nmadd[.]	4	31	FPU	1-1-1	—
ps_nmsub[.]	4	30	FPU	1-1-1	—
ps_res[.]	4	24	FPU	2-1-1	—

Table 6-7. Floating-Point Instructions (Page 3 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
ps_rsqste [.]	4	26	FPU	2-1-1	—
ps_sel [.]	4	23	FPU	1-1-1	—
ps_sub [.]	4	20	FPU	1-1-1	—
ps_sum0 [.]	4	10	FPU	1-1-1	—
ps_sum1 [.]	4	11	FPU	1-1-1	—
fsub [.]	63	20	FPU	1-1-1	—
mcrfs	63	64	FPU	1-1-1	Execution
mffs [.]	63	583	FPU	1-1-1	Execution
mtfsb0v	63	70	FPU	3	—
mtfsb1 [.]	63	38	FPU	3	—
mtfsfi [.]	63	134	FPU	3	—
mtfsf [.]	63	711	FPU	3	—

Table 6-8 shows load and store instruction latencies. Pipelined load/store instructions are shown with cycles of total latency and throughput cycles separated by a colon.

Table 6-8. Load and Store Instructions (Page 1 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
dcbf	31	86	LSU	3:5 ¹	Execution
dcbi	31	470	LSU	3:3 ¹	Execution
dcbst	31	54	LSU	3:5 ¹	Execution
dcbt	31	278	LSU	2:1	—
dcbtst	31	246	LSU	2:1	—
dcbz	31	1014	LSU	3:6 ^{1, 2}	Execution
dcbz_l	4	1014	LSU	3:6 ¹	Execution
eciwx	31	310	LSU	2:1	—
ecowx	31	438	LSU	2:1	—
icbi	31	982	LSU	3:4 ¹	Execution
lbz	34	—	LSU	2:1	—
lbzu	35	—	LSU	2:1	—
lbzux	31	119	LSU	2:1	—
lbzx	31	87	LSU	2:1	—
lfd	50	—	LSU	2:1	—

Note:

1. For cache operations, the first number indicates the latency in finishing a single instruction; the second indicates the throughput for back-to-back cache operations. Throughput can be larger than the initial latency as more cycles might be needed to complete the instruction to the cache, which stays busy keeping subsequent cache operations from executing.
2. The throughput number of 6 cycles for **dcbz** assumes it is to nonglobal (M = '0') address space. For global address space, throughput is at least 11 cycles.
3. Load/store multiple/string instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where *n* is the number of words accessed by the instruction.

Table 6-8. Load and Store Instructions (Page 2 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
lfd	51	—	LSU	2:1	—
lfdx	31	631	LSU	2:1	—
lfdx	31	599	LSU	2:1	—
lfs	48	—	LSU	2:1	—
lfsu	49	—	LSU	2:1	—
lfsux	31	567	LSU	2:1	—
lfsx	31	535	LSU	2:1	—
lha	42	—	LSU	2:1	—
lhau	43	—	LSU	2:1	—
lhaux	31	375	LSU	2:1	—
lhax	31	343	LSU	2:1	—
lhbrx	31	790	LSU	2:1	—
lhz	40	—	LSU	2:1	—
lhzu	41	—	LSU	2:1	—
lhzux	31	311	LSU	2:1	—
lhzx	31	279	LSU	2:1	—
lmw	46	—	LSU	$2 + n^2$	Completion, execution
lswi	31	597	LSU	$2 + n^3$	Completion, execution
lswx	31	533	LSU	$2 + n^3$	Completion, execution
lwarx	31	20	LSU	3:1	Execution
lwbrx	31	534	LSU	2:1	—
lwz	32	—	LSU	2:1	—
lwzu	33	—	LSU	2:1	—
lwzux	31	55	LSU	2:1	—
lwzx	31	23	LSU	2:1	—
psq_l	56	—	LSU	3:1	—
psq_lu	57	—	LSU	3:1	—
psq_lux	4	38	LSU	3:1	—
psq_lx	4	6	LSU	3:1	—
psq_st	60	—	LSU	2:1	—
psq_stu	61	—	LSU	2:1	—
psq_stux	4	39	LSU	2:1	—
psq_stx	4	7	LSU	2:1	—

Note:

- For cache operations, the first number indicates the latency in finishing a single instruction; the second indicates the throughput for back-to-back cache operations. Throughput can be larger than the initial latency as more cycles might be needed to complete the instruction to the cache, which stays busy keeping subsequent cache operations from executing.
- The throughput number of 6 cycles for **dcbz** assumes it is to nonglobal (M = '0') address space. For global address space, throughput is at least 11 cycles.
- Load/store multiple/string instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where n is the number of words accessed by the instruction.

Table 6-8. Load and Store Instructions (Page 3 of 3)

Mnemonic	Primary	Extended	Unit	Cycles	Serialization
stb	38	—	LSU	2:1	—
stbu	39	—	LSU	2:1	—
stbux	31	247	LSU	2:1	—
stbx	31	215	LSU	2:1	—
stfd	54	—	LSU	2:1	—
stfdu	55	—	LSU	2:1	—
stfdux	31	759	LSU	2:1	—
stfdx	31	727	LSU	2:1	—
stfiwx	31	983	LSU	2:1	—
stfs	52	—	LSU	2:1	—
stfsu	53	—	LSU	2:1	—
stfsux	31	695	LSU	2:1	—
stfsx	31	663	LSU	2:1	—
sth	44	—	LSU	2:1	—
sthbrx	31	918	LSU	2:1	—
sthu	45	—	LSU	2:1	—
sthux	31	439	LSU	2:1	—
sthx	31	407	LSU	2:1	—
stmw	47	—	LSU	$2 + n^3$	Execution
stswi	31	725	LSU	$2 + n^3$	Execution
stswx	31	661	LSU	$2 + n^3$	Execution
stw	36	—	LSU	2:1	—
stwbrx	31	662	LSU	2:1	—
stwcx.	31	150	LSU	8:8	Execution
stwu	37	—	LSU	2:1	—
stwux	31	183	LSU	2:1	—
stwx	31	151	LSU	2:1	—
tlbie	31	306	LSU	$3:4^1$	Execution

Note:

1. For cache operations, the first number indicates the latency in finishing a single instruction; the second indicates the throughput for back-to-back cache operations. Throughput can be larger than the initial latency as more cycles might be needed to complete the instruction to the cache, which stays busy keeping subsequent cache operations from executing.
2. The throughput number of 6 cycles for **dcbz** assumes it is to nonglobal (M = '0') address space. For global address space, throughput is at least 11 cycles.
3. Load/store multiple/string instruction cycles are represented as a fixed number of cycles plus a variable number of cycles, where n is the number of words accessed by the instruction.

7. Memory Subsystem Operation

This section describes the Espresso memory subsystem, including the memory interface unit (MIU) in each of the three cores, the core interface unit (CIU) that accepts memory requests from the cores, and the bus interface unit (BIU) that forwards requests from the CIU to the 60Xe bus.

Note: The signals: $\overline{CI}$, $\overline{DBWO}$, $\overline{ARTRY}$, $\overline{DRTRY}$, and $\overline{WT}$ are not supported on the IBM Espresso RISC Microprocessor.

7.1 Memory Interface Unit

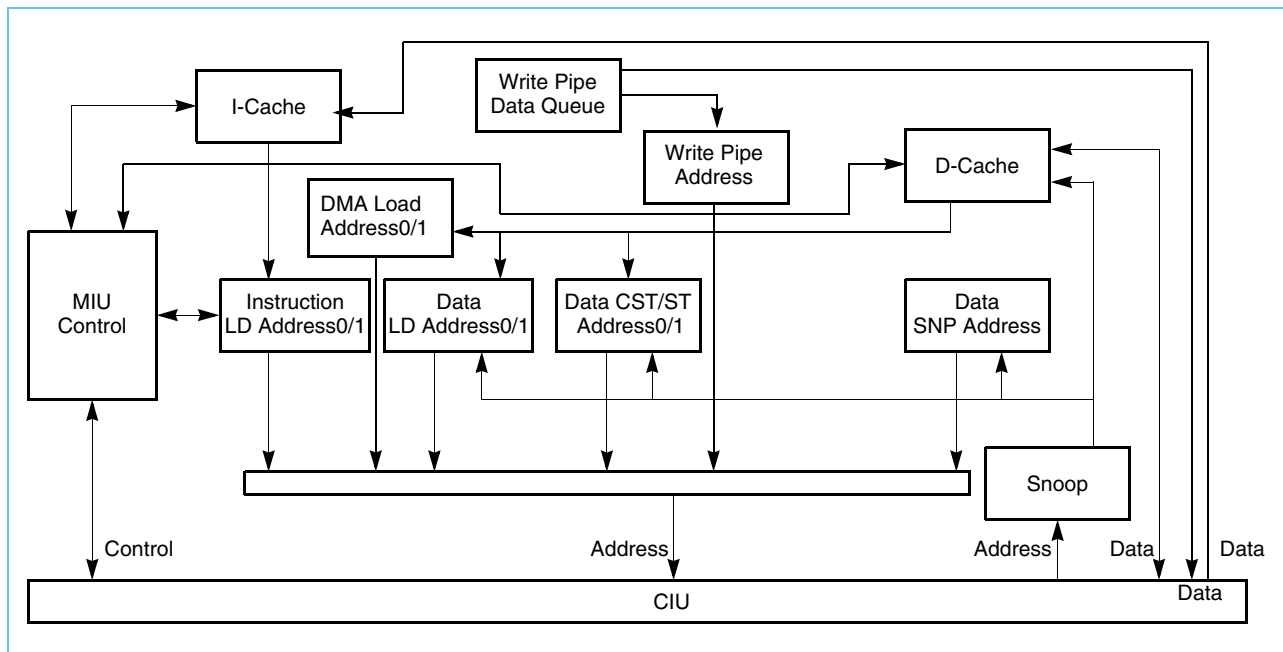
The memory interface buffers bus requests from the caches and forwards them to the CIU for intervention or to be forwarded to the bus. It includes address register queues, prioritizing logic, and bus control logic. It captures snoop addresses for snooping in the cache and in the address register queues. It also snoops for reservations and holds the touch load address for the cache. All data storage for the address register buffers (load and store data buffers) are located in the cache section. The data buffers are considered temporary storage for the cache and not part of the memory interface.

The general functions and features of the memory interface are as follows:

- Address register buffers that include the following:
 - One 64-byte (pair of 32-byte sectors) instruction load address buffer
 - DMA load address buffer
 - Write pipe address buffer
 - One 64-byte (one pair of 32-byte sectors) data load address buffer
 - Two 32-byte L1 data castout or store address buffers (shared with the write gather pipe)
 - One 32-byte snoop copy-back address buffer (the associated data block buffer located in the cache)
 - Reservation address buffer for snoop monitoring
 - One 64-byte (pair of 32-byte sectors) L2 castout address buffer
- Pipeline collision detection for data cache buffers
- Reservation address snooping for **lwarx/stwcx**. instructions
- Address pipelining programmable from two deep (one-level of pipelining) to four deep
- Load ahead of store capability

A conceptual block diagram of the memory interface is shown in *Figure 7-1 Memory Interface Address Buffers* on page 220. The address register queues in the figure hold transaction requests that the memory interface can issue on the bus independently of the other requests. The memory interface can be programmed to support from two to four transactions operating on the bus at any given time through the use of address pipelining.

Figure 7-1. Memory Interface Address Buffers



The memory interface prioritizes requests for bus operations from the caches, and forwards to the CIU. It includes address register queues and prioritization logic. The memory interface latches snoop addresses for snooping in the data cache and in the address register queues, and for reservations controlled by the Load Word and Reserve Indexed (**lwarx**) and Store Word Conditional Indexed (**stwcx.**) instructions, and maintains the touch load address for the cache. The interface allows pipelining depth to be programmed up to a depth of four; that is, with certain restrictions discussed later, there can be four outstanding transactions at any given time. Accesses are prioritized with load operations preceding store operations.

Instructions are automatically fetched from the memory system into the instruction unit where they are dispatched to the execution units at a peak rate of two instructions per clock. Conversely, load and store instructions explicitly specify the movement of operands to and from the integer and floating-point register files and the memory system.

When Espresso encounters an instruction or data access, it calculates the logical address (effective address in the architecture specification) and uses the low-order address bits to check for a hit in the on-chip, 32 KB instruction and data caches.

During cache lookup, the instruction and data memory management units (MMUs) use the higher-order address bits to calculate the virtual address from which they calculate the physical address (real address in the architecture specification). The physical address bits are then compared with the corresponding cache tag bits to determine if a cache hit occurred in the L1 instruction or data cache. If the access misses in the corresponding cache, the physical address is used to access the L2 cache tags (if the L2 cache is enabled). If no match is found in the L2 cache tags, the physical address is used to access system memory.

In addition to the loads, stores, and instruction fetches, Espresso performs hardware table search operations following TLB misses, L2 cache cast-out operations when least-recently used cache lines are written to memory after a cache miss, and cache-line snoop push-out operations when a modified cache line experiences a snoop hit from another bus master.

Figure 1-2 Espresso Core Block Diagram on page 23 shows the address path from the execution units and instruction fetcher, through the translation logic to the caches and bus interface logic.

Espresso uses separate address and data buses and a variety of control and status signals to perform reads and writes. The address bus is 32 bits wide and the data bus is 64 bits wide between each core and the CIU.

7.1.1 Operation of the Instruction and Data L1 Caches

Espresso provides independent instruction and data L1 caches. Each cache is a physically-addressed, 32 KB cache with 8-way set associativity. Because the data cache on Espresso is an on-chip, write-back primary cache, the predominant type of transaction for most applications is burst-read memory operations, followed by burst-write memory operations and single-beat (noncacheable or write-through) memory read and write operations. Additionally, there can be address-only operations, variants of the burst and single-beat operations (global memory operations that are snooped, and atomic memory operations, for example), and address retry activity (for example, when a snooped read access hits a modified line in the cache).

Because the Espresso data cache tags are single ported, simultaneous load or store, DMA access, and snoop accesses cause resource contention. Snoop accesses have the highest priority and are given first access to the tags, unless the snoop access coincides with a tag write, in which case the snoop is retried and must re-arbitrate for access to the cache. Loads or stores that are deferred due to snoop accesses are performed on the clock cycle following the snoop. DMA access has the lowest priority.

Espresso supports a 4-state coherency protocol that supports the modified, exclusive, shared, and invalid (MESI) cache states.

Espresso broadcasts cache control instructions for snooping among the on-chip caches.

Instruction cache lines in Espresso are loaded in four beats of 64 bits each. The burst load is performed as critical doubleword first. The critical doubleword is simultaneously written to the cache and forwarded to the instruction prefetch unit, thus minimizing stalls due to load delays. If subsequent loads follow in sequential order, the instructions are forwarded to the requesting unit as the cache block is written.

Data cache lines in Espresso are loaded into the cache in one cycle for 256 bits. For a cache line load due to the cache miss of a load instruction, the critical doubleword is simultaneously written to the 256 bit line fill buffer and forwarded to the requesting load/store unit. If subsequent loads follow in sequential order, the data is forwarded to the load/store unit as the cache block is written into the cache. For DMA read and data cache cast out, it takes one cycle to read the data out of the cache.

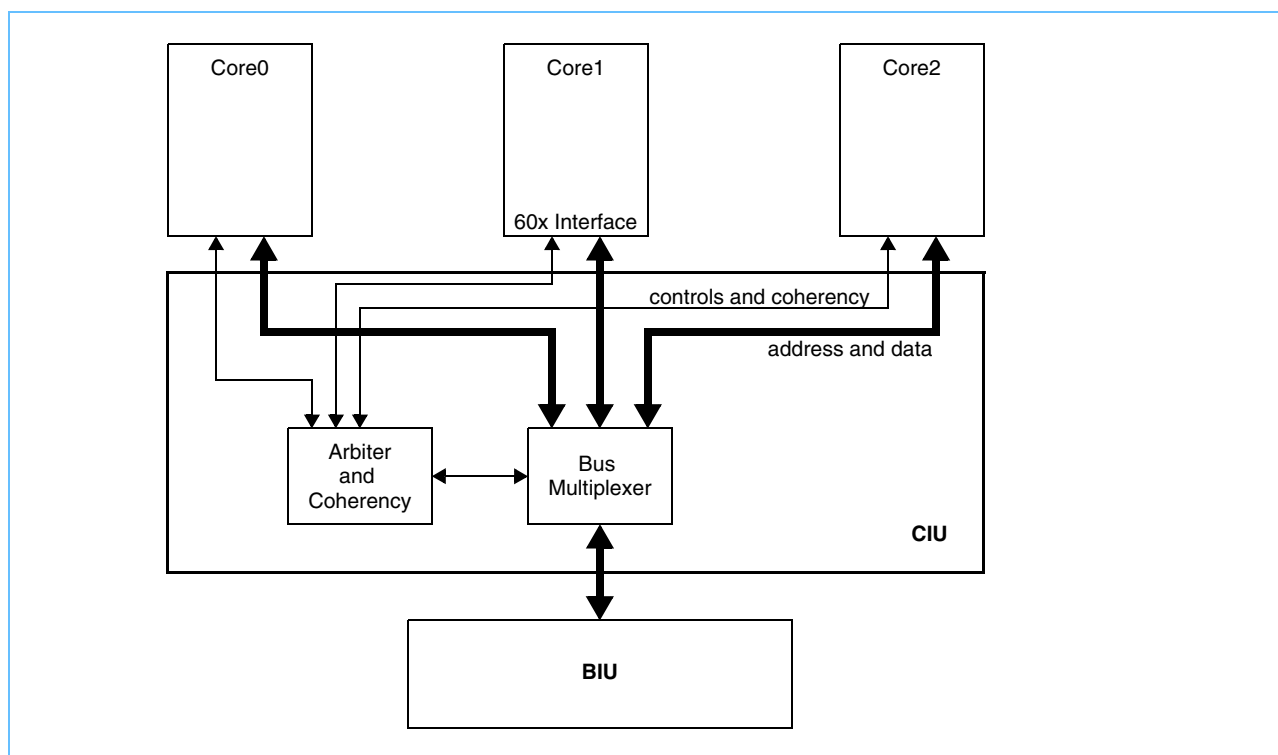
Cache lines are selected for replacement based on a pseudo least-recently-used (PLRU) algorithm. Each time a cache line is accessed, it is tagged as the most-recently-used line of the set. When a miss occurs, and all eight lines in the set are marked as valid, the least recently used line is replaced with the new data. When data to be replaced is in the modified state, the modified data is written into a write-back buffer while the missed data is being read from memory. When the load completes, Espresso then pushes the replaced line from the write-back buffer to the L2 cache (if enabled), or to main memory in a burst write operation.

7.2 Core Interface Unit

The core interface unit (CIU) accepts load and store requests, as well as coherency operations, from the three cores, arbitrates among those requests, broadcasts those requests to the other cores to maintain coherency and for possible intervention, and forwards the requests to the BIU as necessary. The arbitration scheme is round robin, with an additional feature to throttle requests from individual cores to allow software control of bandwidth allocation.

Figure 7-2 shows a block diagram of the CIU.

Figure 7-2. CIU Block Diagram



7.2.1 Intervention

The Espresso chip supports intervention among the three cores. The purpose of intervention is to reduce the latency needed to fetch a cache line when it is not present in the requesting core's cache, but is present on-chip in another cache. The following are characteristics of the design:

- Three cores are on the chip, each with the same L1 as in previous designs, plus support for the shared state.
- Each core has a private L2 cache with different signal timings than those in previous designs.
- Coherency states in the L1 and L2 caches are MESI.
- Intervention data can come from the L1 or L2 cache.
- CIU waits for intervention responses before forwarding requests to the bus.
- Possible intervention responses are miss, shared, and modified.
- Only cacheable load, touch, and store instructions generate intervention requests.

In compatibility mode, only core 0 is enabled. The CIU and BIU are bypassed, sending all requests directly to the 60x bus.

7.2.1.1 Cache Coherency States

The cache hierarchy and CIU collectively support the 5-state MERSI cache coherency protocol. The cache hierarchy of each core associates every cache line with one of four states. The R state is implicitly enforced by the CIU. The five states are defined in *Table 7-1*.

Table 7-1. Cache Coherency States

State	Description
M - modified	The cache line is in this cache and only this cache, and is modified with respect to memory.
E - exclusive	The cache line is present in this cache and only this cache, and is not modified with respect to memory.
R - recent	The cache line is present in this cache and might be present in one or more other caches, is not modified with respect to memory, and is the most recent copy of this cache line among all caches.
S - shared	The cache line is present in this cache and might be present in one or more other caches, is not modified with respect to memory, and is not the most recent copy of this cache line among all caches.
I - invalid	The cache line is not present in this cache.

Because the L2 is noninclusive, and the L1 is normally managed using a write-back policy, a given cache line might be in a different state in the L1 than it is in the L2. However, the cache hierarchy (combined L1 and L2) of a given core responds to snoops and intervention requests in a manner consistent with that cache line being in one of the four MESI coherency states.

In the previous design, when a miss occurs in the L2, it allocates the new line as Exclusive at the time of the miss. With intervention, that line might come back in either the E or S states. For this reason, the L2 allocates the new line as Shared at the time of the miss, and in the case of intervention must update its state when the critical word is returned.

When an intervening core supplies intervention data, that data comes from the L1 cache if it contains the line; otherwise, the data comes from the L2 cache.

In compatibility mode, the CIU and BIU are bypassed, and all transactions are sent directly from the MIU to the external bus, without any snooping or intervention among the cores.

7.2.1.2 Snoop Handling

The MEI coherency protocol supported in the previous design has been enhanced to MESI support in the Espresso core. Support for the shared (S) state makes use of an explicitly represented fourth coherency state for each cache line in the L1 and L2 caches. In addition, support for the recent (R) state is approximated through ordering conventions enforced in the CIU among the cores when intervening.

The transaction types that can be generated by the cores, and the handling of those transaction types by the CIU is shown in *Table 7-2*. The write-through (W), cache-inhibited (I), and global or memory coherency required (M) attributes of each transaction are shown in the WIM column. A letter in this column (W, I, or M) indicates that the WIM attributes come from the MMU. A '1' or '0' in this column indicates that the attribute is fixed for the corresponding transaction type.

Note: It is possible to alias multiple effective addresses to a given physical address, and otherwise assign different memory and coherency characteristics to that physical address. However, the same WIM settings must be assigned to a given physical address throughout the system. It is considered a programming error to do otherwise.

The CIU broadcasts a transaction from one core to the other two cores to be snooped. In the “Cache Snoop” column, transactions that are never snooped are indicated. Those Read type transactions that can be intervened are indicated in that column by ‘intervention’, and described in greater detail below. Other transaction types are snooped in the other cores to perform a clean, flush, or kill block. If the M bit is asserted, the transaction is broadcast for snooping. If the M bit is negated, the transaction is forwarded directly to the BIU. However, for the Write-with-Kill transaction type, a “kill reservation” signal is sent to the other cores, independent of the value of M.

Table 7-2. CIU Snoop Handling of Memory Requests (Page 1 of 2)

Instruction	Transaction Type	TT	WIM	CIU Snoop Handling		
				Cache Snoop	Reservation Snoop	I-Cache, TLB Snoop
Reads						
dcbt load	Read	0A	WIM	intervention		
DMA load last	Read	0A	-10	not snooped		
store dcbtst	RWITM	0E	WIM	intervention	kill reservation (MESI only)	
DMA load	RWNITC	0B	-10	not snooped		
L2 prefetch	RWNITC	0B	WIM	intervention		
CI Read	Single-beat read	0A	W1M	clean block		
lwarx	Read atomic	1A	WIM	intervention		
stwcx.	RWITM atomic	1E	WIM	intervention	kill reservation (MESI only)	
Writes						
castout due to: load store dcbz	Write with kill	06	100	not snooped	kill reservation	
push due to: dcbf dcbst	Write with kill	06	000	not snooped	kill reservation	

Table 7-2. CIU Snoop Handling of Memory Requests (Page 2 of 2)

Instruction	Transaction Type	TT	WIM	CIU Snoop Handling		
				Cache Snoop	Reservation Snoop	I-Cache, TLB Snoop
DMA store	Write with flush	02	-10	not snooped		
CI store WT store	Write with flush	02	WIM	flush block	kill reservation	
WP store	Write with flush	02	000	not snooped		
stwcx.	Write with flush atomic	12	WIM	flush block	kill reservation (any address)	
Cache control						
dcbst	Clean	00	WIM	clean block		
dcbf	Flush	04	WIM	flush block		
dcbi	Kill	0C	WIM	kill block	kill reservation	
dcbz	Kill	0C	00M	kill block	kill reservation	
icbi	icbi	0D	111			kill L1 set, snooped for pending icbi
tlbie	tlbie	18	111			kill TLB set, snooped for pending tlbie
Synchronization						
eieio	eieio	10	111	not snooped		
sync	sync	08	111			snooped for pending icbi
tlbsync	tlbsync					'snooped' for pending tlbie using the internal tlbsync signal
Other						
eciwx	eciwx	1C	000	not snooped		
ecowx	ecowx	14	000	not snooped		
lwarx	lwarx	01	00M	not broadcast		

Operations within the Espresso core generate the same memory transactions as in the previous design, with the following exceptions. The snoop response to each of these transaction types is new to the Espresso core.

- A **tlbie** instruction generates a TLBIE transaction that is snooped by the other cores, and the TBL congruency set is invalidated.
- An **icbi** instruction generates an ICBI transaction that is snooped by the other cores, and the L1 I-Cache congruency set is invalidated.
- The **tlbsync** instruction behaves the same as in the previous design, waiting for the internal tlbsync signal to be negated before completing. The tlbie_pending signals from the other two cores are combined to drive the internal tlbsync input signal to the core that executed the **tlbsync** instruction to provide the appropriate synchronization behavior.

7.2.1.3 Intervention Responses

The response to an intervention request is either miss, shared, or modified. The CIU combines the responses from the two snooped cores to determine the combined intervention response. *Table 7-3* describes the various valid combinations of snoop responses for a given cache line, along with the combined responses to an intervention request. *Table 7-4* is a similar table for the coherency paradox cases. When the CIU detects a paradox case, it sets the paradox bit.

Table 7-3. Intervention Response to a Request from Core A (Normal Coherency Cases)

Core B Response	Core C Response	Combined Response	CIU Action
miss	miss	miss	forward to bus
miss	shared	shared	shared intervention
miss	modified	modified	modified intervention
shared	shared	shared	shared intervention

Table 7-4. Intervention Response to a Request from Core A (Paradoxical Coherency Cases)

Core B Response	Core C Response	Combined Response	Requester Action	Paradox Next
share	modified	modified	modified intervention	Both cores push data CIU selects modified intervention data
modified	modified	modified	modified intervention	Both cores push data CIU selects intervention data from one of the cores

Several aspects of the intervention responses are as follows:

- A response of shared or modified is accompanied by the intervened data.
- When multiple cores respond with intervention data, the CIU selects the data from one of the cores and kills the second push. This CIU selection process provides a function similar to the use of the R bit in the MERSI protocol by uniquely identifying one cache as being the intervener among caches that share a copy of a cache line.
- When one cache responds modified, all other caches should respond miss. A second cache responding shared or modified in this case corresponds to a coherency paradox. The CIU handles such paradoxes by selecting the modified intervener, if there is only one, and otherwise simply selects one of the cores as the intervener.

- Due to deallocations associated with line replacements, a cache line marked S might be removed from a cache, resulting in a cache line marked S in another cache being the only cached copy of the line. This causes an occasional extraneous RWITM operation on the bus (when modifying a cache line marked S if it is the only cached copy). The related situation for cache lines marked R in a strict MERSI implementation can lead to an occasional missed intervention opportunity (when there is no cache line marked R associated with one that is marked S). However, in this implementation, all cache hits are reported, so that the CIU can select an intervener from among them.

The instructions in *Table 7-5* generate corresponding data fetch transactions when they miss in the local caches that in turn initiate intervention requests.

Table 7-5. Instructions that Generate Intervention Requests

Instruction Type	Data Transaction
load	READ
lwarx	READA
store	RWITM
dcbt	READ
dcbtst	READ

Table 7-6 describes the state changes of the requesting and intervening caches associated with READ, READA, RWITM, and RWITMA transactions. For the RWITM transactions, the received cache line is first marked E, and then when the store instruction executes, the state is changed to M.

Table 7-6. Coherency State Changes for Intervention Operations

Operation	Intervener Current State	Intervention Response	Intervener New State	Requester New State	Bus Transaction
READ[A]	I	miss	-	E	Read[A]
READ[A]	E, S	shared	S	S	-
READ[A]	M	modified	I	E	Write with kill
RWITM[A]	I	miss	-	E then M	RWITM[A]
RWITM[A]	E, S	shared	I	E then M	-
RWITM[A]	M	modified	I	E then M	Write with kill

7.3 Bus Interface Unit Overview

The bus interface unit in Espresso receives bus requests from the CIU that arbitrates requests coming from the three cores. The BIU forwards those requests to the bus in the order received, but supports responses from the bus arbiter that come back out of order. This data reordering capability is one of several extensions to the bus protocol supported in the previous design. The others are the use of a 128-bit wide data bus and the removal of certain idle cycles for both the address and data tenures.

The previous design supported a maximum bus pipeline depth of four (the number of outstanding transactions on the bus). The Espresso continues to make bus requests until as many as 16 transactions are outstanding.

8. L2 Cache, Locked D-Cache, DMA, and Write Gather Pipe

This section describes the Espresso microprocessor implementation of the L2 cache, L1 D-Cache partition, direct memory access (DMA), and write gather pipe.

8.1 L2 Cache Overview

The Espresso chip has 3 MB of embedded dynamic random access memory (eDRAM) to be used for the L2 cache by the three PowerPC cores. The 3 MB is partitioned in normal use into a 512 KB block for core 0, a 2 MB block for core 1, and a 512 KB block for core 2. Each of these caches is 4-way set-associative and 2-sectored. For the purpose of a compatibility mode, a 512 KB block can be configured as a 256 KB, 2-way set-associative cache by preventing allocation in ways 2 and 3.

The L2 controller, contained in each of the three cores, supports the L2 cache configurations shown in *Table 8-1*.

Table 8-1. Supported Cache Configurations

	Set Associativity	Sectoring	Number of Sets
2 MB	4-way	2-sectored	8192
512 KB	4-way	2-sectored	2048
256 KB	2-way	2-sectored	2048

The caches are physically addressed. The 32-bit physical address is partitioned as shown in *Table 8-2*.

Table 8-2. Address Bit Assignments for Different Cache Configurations

	Tag	Set Index Bits	Sector Bit	qw Select Bit	Offset
2 MB	0:12	13:25	26	27	28:31
512 KB	0:14	15:25	26	27	28:31
256 KB	0:14	15:25	26	27	28:31

8.1.1 L2 Cache Tag Array

To control the three different cache configurations, the tag array must have the capacity to maintain the tags and associated information for any of those configurations. Since cores 0 and 2 can only support the 512 KB and 256 KB cache configurations, the L2 tag arrays for these cores have 2K entries. Core 1 requires a tag array containing 8K entries to support the 2 MB L2 configuration. For each of the 2K or 8K sets, the tag array contains LRU bits and an entry for each of the four ways. That entry contains the tag and two groups of coherency bits, one for each sector. The maximum tag length, used for the 512 KB cache, is 15 bits. A pseudo-LRU (PLRU) algorithm requires 3 bits for a 4-way set-associative cache. To support MESI coherency requires 2 bits to encode one of the four coherency states. Thus, each tag array entry contains the following:

- 15 tag bits for each of 4 ways
- 3 LRU bits
- 2 MESI bits for each of 4 ways for each of 2 sectors

for a total of 79 bits for each of the 2K or 8K entries. The total capacity of the L2 tag array for the 512 KB caches is therefore 2K entries $\times$ 79 bits = 20 KB, while that for the 2 MB cache is 8K entries $\times$ 79 bits = 80 KB.

In a given tag lookup, all four ways of a given set must be compared with the tag field of the physical address. The tag array is organized as 2K or 8K entries of 79 bits each. The tag array is addressed by the 11 or 13 bits of the set index to select an entry, plus sector bits to choose which of the two groups of MESI bits apply to the current lookup. The tag array layout for the 2 MB L2 cache configuration is shown in *Table 8-3* and *Table 8-4*. The corresponding layout for the 512 KB L2 cache is identical. For the 256 KB L2 cache, the layout is the same, but the tag 2 and tag 3 fields and corresponding MESI bits are unused in that case.

Table 8-3. L2 Tag Array Entry Layout

Tag0	Tag1	Tag2	Tag3	00	01	10	11	20	21	30	31	LRU
0:14	15:29	30:44	45:59	60	62	64	66	68	70	72	74	76:78

Table 8-4. L2 Tag Array Entry Bit Description

Bits	Field
0:14	tag0
15:29	tag1
30:44	tag2
45:59	tag3
60:61	MESI_00
62:63	MESI_01
64:65	MESI_10
66:67	MESI_11
68:69	MESI_20
70:71	MESI_21
72:73	MESI_30
74:75	MESI_31
76:78	LRU

When a load or store access to the L2 misses, the corresponding cache line must be brought into the cache from memory. In this case, one of the ways in the set is selected for replacement with the new cache line, using the PLRU replacement algorithm. This algorithm is described in *Table 8-5*. When the L2 cache is configured as a 256 KB, 2-way set associative cache, the B2 bit is ignored (assumed 0), and replacement is based on the value of B0, as shown in the table.

Table 8-5. L2 PLRU Replacement Algorithm

If the PLRU bits are:				The way replaced is:
B2	0	B0	0	0
	0		1	1
	1	B1	0	2
	1		1	3

Whenever a load or store hits in the L2, the way that is accessed is marked as the most recently used (MRU) way. The update rules for the three LRU bits in such cases are described in *Table 8-6*. The update rules in this table are also used to modify the LRU bits when a new cache line is allocated due to a replacement, to make the way selected for replacement the MRU way

Table 8-6. L2 PLRU Bit Update Rules for Loads and Stores

If the current access is to way:	Then the PLRU bits are changed to:		
	B2	B0	B1
0	1	1	X
1	1	0	X
2	0	X	1
3	0	X	0

In the case of L2 accesses by cache control operations, the way that is accessed is marked as the least-recently used (LRU) way. The update rules for the 3 LRU bits for cache operations are described in *Table 8-7*.

Table 8-7. L2 PLRU Bit Update Rules for Cache Operations

If the current access is to way:	Then the PLRU bits are changed to:		
	B2	B0	B1
0	0	0	X
1	0	1	X
2	1	X	0
3	1	X	1

When a cache line is filled due to an intervention from another core, the LRU is updated as if the linefill were a normal load or store access. That is, if the linefill hits one of the ways, as it will almost always do, it marks that way as the MRU, as described in *Table 8-6*. If the linefill misses, due to an invalidate that occurs between the time the line was allocated and when the data is written, the current LRU way will be selected, according to *Table 8-5*, and that way will be made the MRU, according to *Table 8-6*.

8.1.2 Cache Coherency

Because the L2 is noninclusive, and the L1 is normally managed using a write-back policy, a given cache line might be in a different state in the L1 cache than it is in the L2 cache. However, the cache hierarchy (combined L1 and L2) of a given core responds to snoops and intervention requests in a manner consistent with that cache line being in one of the four MESI coherency states. The combinations of L1 and L2 coherency states for a given cache line, and its corresponding collective coherency state are listed in *Table 8-8*. An asterisk in the collective state column of that table indicates a combination that is not expected to occur. Also listed in that table is the identity of the intervening cache in each case. While the L1 cache will intervene with data whenever it has it, the L2 cache will only intervene if the L1 cannot supply the data.

Table 8-8. L1 - L2 Combined Coherency State

L1 State	L2 State	Collective State	Intervener
I	I	I	-
I	E	E	L2
I	S	S	L2
I	M	M	L2
E	I	E	L1
E	E	E	L1
E	S	E	L1
E	M	E*	L1
S	I	S	L1
S	E	S*	L1
S	S	S	L1
S	M	S*	L1
M	I	M	L1
M	E	M	L1
M	S	M	L1
M	M	M	L1

The L2 cache responds to requests from the LSU as described in *Table 8-9*.

Table 8-9. L2 Data Cache State Transitions in Response to L1/LSU Operations (Page 1 of 2)

Request Type	Operation	WT	CI	L1 State	Current L2 State	Next L2 State	Action	Bus Transaction Type
Read	load; dcbt; dcbtst	-	0	I	I	E, S, M	Castout victim	Write-with-Kill
					E, S, M	Same	Request from CIU	Read
		-	1	I, E, S, M	I, E, S, M	Same	Return line to L1	-
Read atomic	lwarx	-	0	I	I	E, S, M	Castout victim	Write-with-Kill
					E, S, M	Same	Request from CIU	ReadA
		-	1	I, E, S, M	I, E, S, M	Same	Return line to L1	-
		-	1	I, E, S, M	I, E, S, M	Same	Request from CIU	Single-beat ReadA

Table 8-9. L2 Data Cache State Transitions in Response to L1/LSU Operations (Page 2 of 2)

Request Type	Operation	WT	CI	L1 State	Current L2 State	Next L2 State	Action	Bus Transaction Type	
Write-with-kill	Replacement castout	0	0	M	I	M	Castout victim	Write-with-Kill	
						Write to cache	–		
	Snoop push; dcbf ; dcbst	0	0	M	E, S, M	M	Write to cache	–	
					I, E, S, M	I	Kill line	–	
RWITM	Store	0	0	I, S	I	E, S, M	Castout victim	Write-with-Kill	
							Request from CIU	RWITM	
							Write to cache	–	
							Return line to L1	–	
					E, M	Same	Return line to L1	–	
					S	E, S, M	Request from CIU	RWITM	
							Write to cache	–	
							Return line to L1	–	
Write-with-flush	Store	–	1	I, E, S, M			I, E, S, M	Same	Write to memory
					I	Same	Write to memory	Single-beat write-with-flush	
		1	0	I, E, S, M	E, S	E	Write to cache	–	
							Write to memory	Single-beat write-with-flush	
M					Same	Write to cache	–		
						Write to memory	Single-beat write-with-flush		
Write-with-flush atomic		stwcx.	–	–	I, E, S, M	I, E, S	I	Write to memory	Single-beat write-with-flush-atomic
						M	M	Write to cache	–
	Write to memory							Single-beat write-with-flush-atomic	
Flush	dcbf	–	–	I, E, S	I, E, S	I	Flush other copies	Flush	
					M	I	Write to memory	Write-with-kill	
Clean	dcbst	–	–	I, E, S	I, E, S	Same	Clean other copies	Clean	
					M	E	Write to memory	Write-with-kill	
Kill	dcbi	–	–	I, E, S, M	I, E, S, M	I	Kill other copies	Kill	
	dcbz	–	–	I, S	I, S	Same	Kill other copies	Kill	

8.1.2.1 Tag Updates

The L2 cache on previous designs preallocates all L2 misses as Exclusive before the data from memory is returned. Espresso preallocates like earlier designs, but also implements a new function called "MESI injection" to allocate on reload under certain conditions. The preallocation is the Shared state. Preallocation is always performed first, with MESI injection performed second if the data is from an intervening core. If no core intervenes, the reload data comes from external memory (BIU) and its final state of the cache line in the L2 is the preallocated state. When a core intervenes, the requesting core snoops the intervention transaction and updates the L2 cache line state to M, E, or S. This is referred to as MESI injection.

8.1.3 Dataflow

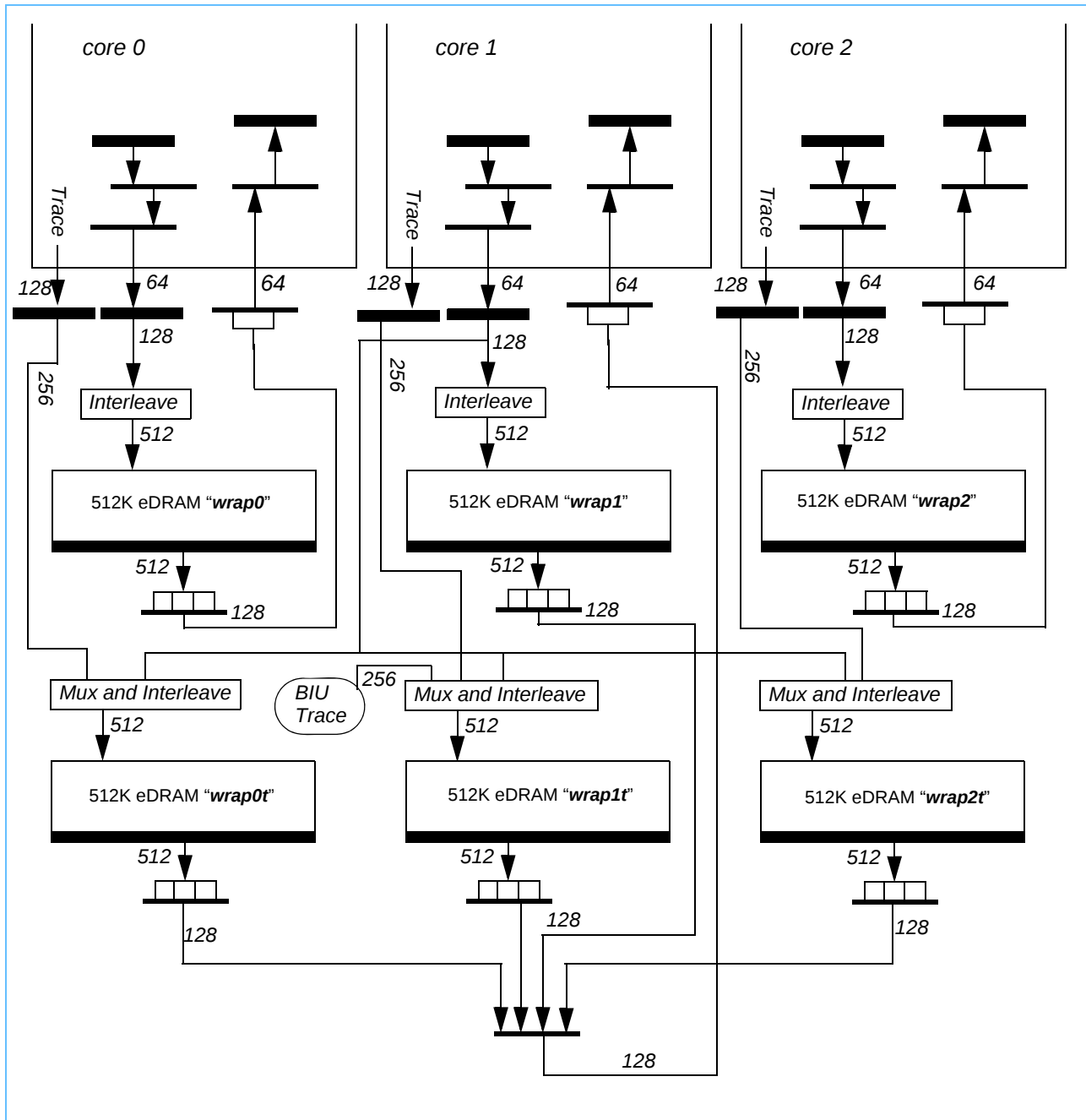
Figure 8-1 shows the dataflow between the three cores and the L2 data arrays. Each of the six data arrays shown is organized as 8K word lines by 512 bits. The data paths between the cores and L2s are all 64 bits wide, as are the core registers. The array on the upper left is the 512 KB data array for core 0. The array on the upper right is the 512 KB data array for core 2. The remaining four arrays comprise the 2 MB data array for core 1.

For L2 writes, the 128-bit wide L2 input register receives 64 bits of data on successive cycles, and then writes the entire 128 bits of data in one L2 write cycle. In the case of core 1, only one of the four L2 arrays is selected, according to the high-order set-index bit and the high-order sector bit. A cache line write requires two such write operations.

For L2 reads, an entire 512-bit entry is read, from which 128 bits are selected by the output multiplexer according to the way. A second multiplexer then selects the high-order or low-order 64 bits to transfer back to the core on one cycle, and the other 64 bits to transfer back to the core on the succeeding cycle. Two such reads are required to read an entire cache line. In the case of core 1, only one of the four L2 data arrays is selected and its corresponding data routed to the core, according to the value of the two high-order set-index bits.

The address bits for the L2 cache data arrays are multiplexed in the core from the various request latches. For cores 0 and 2, a 15-bit address is needed to select a 16-byte value from a 512 KB array. The 13-bit data array address itself comes from the 11-bit set-index address, plus the sector bit and the qw select bit. The other two address bits are the way select bits that control the output multiplexer of the array. For core 1, a 17-bit address is needed to select a 16-byte value from a 2 MB array. The 13-bit data array address in this case comes from the low-order 11 bits of the set address, plus the sector bit and the qw select bit. Two additional bits are the way-select bits that control the output multiplexer of each array, and two other address bits are the high-order set-index bits, which select one of the four data arrays to access, and control the array select multiplexer.

Figure 8-1. L2 Core Dataflow



8.1.4 L2 Cache Operation

The L2 cache can be configured to fetch one or two 32-byte sectors whenever a load miss occurs. In the following subsection, the overall operation of the L2 cache is described, assuming the 32-byte fetch mode for details that depend on the number of sectors fetched. Subsequent sections describe the differences in this overall behavior when configured for 64-byte fetch mode.

8.1.4.1 32-Byte Fetch Mode

The Espresso L2 cache is a combined instruction and data cache that receives memory requests from both L1 instruction and data caches independently. The L1 requests are generally the result of instruction fetch misses, data load or store misses, write-through operations, or cache management instructions. Each L1 request generates an address lookup in the L2 tags. If a hit occurs, the instructions or data are forwarded to the L1 cache. A miss in the L2 tags causes the L1 request to be forwarded to the CIU. The cache block received from the bus is forwarded to the L1 cache immediately, and is also loaded into the L2 cache with the tag marked valid and unmodified. If the cache block loaded into the L2 causes a new tag entry to be allocated and the current tag entry is marked valid modified, the modified sectors of the tag to be replaced are castout from the L2 cache to the bus.

At any given time the L1 instruction cache can have one instruction fetch request, and the L1 data cache can have one load and two stores requesting L2 cache access. The L2 cache also services snoop requests from the bus. When there are multiple pending requests to the L2 cache, snoop requests have highest priority, followed by data load and store requests (serviced on a first-in, first-out basis). Instruction fetch requests have the lowest priority in accessing the L2 cache when there are multiple accesses pending.

If read requests from both the L1 instruction and data caches are pending, the L2 cache can perform hit-under-miss and supplies the available instruction or data while a bus transaction for the previous L2 cache miss is performed. Under the control of HID4[L2MUM], the L2 cache can be configured to support a 2-deep, miss-under-miss (MUM) capability. When HID4[L2MUM] = '0', the L2 cache does not support miss-under-miss, and the second instruction fetch or data load stalls until the bus operation resulting from the first L2 miss completes. When HID4[L2MUM] = '1', an instruction fetch that misses after a missed data load, or a data load that misses after a missed instruction fetch, will immediately be forwarded to the CIU.

All requests to the L2 cache that are marked cacheable (even if the respective L1 cache is disabled or locked) cause tag lookup and will be serviced if the instructions or data are in the L2 cache. Burst and single-beat read requests from the L1 caches that hit in the L2 cache are forwarded instructions or data, and the L2 LRU bit for that tag is updated. Burst writes from the L1 data cache due to a castout or replacement copyback are written only to the L2 cache, and the L2 cache sector is marked modified.

If the L2 cache is configured as write-through, the L2 sector is marked unmodified, and the write is forwarded to the bus. If the L1 castout requires a new L2 tag entry to be allocated and the current tag is marked modified, any modified sectors of the tag to be replaced are cast out of the L2 cache to the bus.

Single-beat read requests from the L1 caches that miss in the L2 cache do not cause any state changes in the L2 cache and are forwarded on the CIU. Cacheable single-beat store requests marked copy-back that hit in the L2 are allowed to update the L2 cache sector, but do not cause L2 cache sector allocation or deallocation. Cacheable, single-beat store requests that miss in the L2 are forwarded to the bus. Single-beat store requests marked write-through (through address translation or through the configuration of L2CR[L2WT]) are written to the L2 cache if they hit and are written to the bus independent of the L2 hit/miss status. If the store hits in the L2 cache, the modified/unmodified status of the tag remains unchanged. All requests to the L2 cache that are marked cache-inhibited by address translation (through either the MMU or by default WIMG configuration) bypass the L2 cache and do not cause any L2 cache tag state change.

The execution of the **stwcx.** instruction results in a single-beat write request to the L2 from the L1 data cache. These single-beat writes are processed by the L2 cache according to hit/miss status, L1 and L2 write-through configuration, and reservation-active status. If the address associated with the **stwcx.** instruction misses in the L2 cache or if the reservation is no longer active, the **stwcx.** instruction bypasses the L2 cache and is forwarded to the CIU. For proper coherency, the cache line addressed by the **lwarx/stwcx.** pair should be in a cache-inhibited address space. Alternatively, a **dcbf** instruction can be executed before the **stwcx.** (see *Table 2-38. Memory Synchronization Instructions (UISA)* on page 103). In either case, the **stwcx.** does not hit in the L2 cache.

L1 cache-block-push operations generated by the execution of **dcbf** and **dcbst** instructions write through to the CIU and invalidate the L2 cache sector if they hit. The execution of **dcbf** and **dcbst** instructions that do not cause a cache-block-push from the L1 cache are forwarded to the L2 cache to perform a sector invalidation and/or push from the L2 cache to the bus as required. If the **dcbf** and **dcbst** instructions do not cause a sector push from the L2 cache, they are forwarded to the CIU for address-only broadcast if **HID0[ABE]** is set to '1'.

The L2 flush mechanism is similar to the L1 data cache flush mechanism. L2 flush requires that the entire L1 data cache be flushed before flushing the L2 cache. Also, interrupts must be disabled during the L2 flush so that the LRU algorithm does not get disturbed. The L2 can be flushed by executing uniquely addressed load instructions to each of the 32 byte blocks of the L2 cache. This requires a load to each of the two sets (2-way, set associative) of the 32-byte block (sector) within each 64-byte line of the L2 cache. The loads must not hit in the L1 cache to effect a flush of the L2 cache.

The **dcbi** instruction is always forwarded to the L2 cache and causes a sector invalidation if a hit occurs. The instruction is also forwarded to the CIU for broadcast if **HID0[ABE]** is set to '1'. The **icbi** instruction invalidates only L1 cache blocks and is never forwarded to the L2 cache.

Any **dcbz** instructions marked global do not affect the L2 cache state. If an instruction hits in the L1 and L2 caches, the L1 data cache block is cleared and the instruction completes. If an instruction misses in the L2 cache, it is forwarded to the CIU for broadcast. Any **dcbz** instructions that are marked nonglobal act only on the L1 data cache without reference to the state of the L2.

The **dcbz_I** is not forwarded to the L2 cache. The **sync** and **eieio** instructions bypass the L2 cache and are forwarded to the bus.

8.1.4.2 64-Byte Fetch Mode

The L2 cache is organized as 64-byte lines, each line having an associated real address tag. Every 64-byte line is composed of two 32-byte sectors, each sector having its own valid and modified bit. This organization lends itself to several management schemes, each providing a performance advantage for certain types of application. In 32-byte fetch mode, as previously described, a load (either data or instruction) miss in the L2 cache generates a request to memory for the 32-byte sector containing the data or instruction being requested. In 64-byte fetch mode, both 32-byte sectors of an L2 cache line are requested in response to a load miss.

More specifically, the request for the “critical sector”, which contains the data or instruction that caused the cache miss, is sent to the memory interface unit first, followed by a second request for the other 32-byte sector associated with the same tag. Except for this additional read request in response to a load miss, all other aspects of the L2 cache behavior are the same as in 32-byte fetch mode.

If the critical sector is not in the L2 cache, but the corresponding tag is allocated (and so the other 32-byte sector is resident), the critical sector will be requested from memory, but an extraneous request for the other sector will not be generated. As is the case for the 32-byte fetch mode, a load hit results in the critical sector being forwarded to the appropriate L1 cache, but if the other sector is not resident, it will not be requested from memory. Similarly, a store miss, as from an L1 castout, results in allocation of a 64-byte line, but only the referenced 32-byte sector is written with data, while the other sector is simply marked invalid.

Note: When a cache line contains both instructions and data, any bus transaction that references that line must be snooped by all cores to ensure coherency of all copies. This requires that the M bit be set for the block or page in which the line resides and that HID0[IFEM] be set to allow that M bit to be broadcast when that cache line is fetched as instructions. When the L2 is configured for 64-byte fetch mode (L2 prefetch), this requirement applies for any 64-byte, aligned block of memory that contains both instructions and data.

8.2 Locked L1 Data Cache

Under the control of the HID2[LCE] bit, the L1 data cache can be configured as either a 32 KB normal cache, or as a 16 KB normal cache and a 16 KB locked cache. The locked cache can be explicitly managed, separate from the normal cache. The **dcbz_l** instruction is used to allocate cache lines in the locked cache.

8.2.1 Locked Cache Configuration

At power-on or reset, HID2[LCE] is set to be '0'. The L1 data cache is a 32 KB, 8-way set-associative cache, as described in *Section 3 Espresso Instruction and Data Cache Operation* on page 111. To enable the locked cache, the L1 data cache must first be invalidated. When an **mtspr** instruction sets HID2[LCE] = '1', the data cache is configured as two partitions. The first partition, consisting of ways 0 - 3, is a 16 KB normal cache. The second partition, consisting of ways 4 - 7, is a 16 KB locked cache. The normal cache operates as described in *Section 3 Espresso Instruction and Data Cache Operation*, except that it behaves as a 4-way, set-associative cache. The operation of the locked cache partition is described in the following sections.

Before the HID2[LCE] bit is set, the L1 data cache must be flushed and invalidated. This prevents non-DMA cache lines from being left in the locked side of the L1 data cache, where they are not subject to normal cache management.

Before the HID2[LCE] bit is reset, the L1 data cache must be cleaned up in one of two ways:

1. The cache lines in the locked side of the L1 data cache are not maintained coherently with physical memory. Therefore, they must be invalidated using a sequence of **dcbi** instructions.
2. The cache lines in the unlocked side of the L1 data cache can be flushed from the cache. Then, the entire L1 data cache can be flash invalidated.

8.2.2 Locked Cache Operation

The **dcbz_I** instruction is the only mechanism to allocate a tag for a 32-byte block in the locked cache to be associated with a particular address. There are three methods to de-allocate cache lines in the locked cache:

1. Use the **dcbi** instruction.
2. Use the **dcbf** instruction.
3. The **dcbz_I** instruction forces cache line replacement by the PLRU algorithm in the locked cache.

The behavior of the cache control instructions follow.

8.2.2.1 *dcbz*

If a **dcbz** instruction misses both the normal cache and the locked cache, then a cache line is allocated from the four ways in the normal cache according to the pseudo-LRU rule. The effect in the L2 and the bus is the same as when $HID2[LCE] = '0'$

If the instruction hits either the locked cache or the normal cache, the cache line is cleared and marked as 'M' and the effect in the L2 and the bus is the same as when $HID2[LCE] = '0'$.

8.2.2.2 *dcbz_I*

If a **dcbz_I** instruction misses both the normal cache and the locked cache, a cache line is allocated from the four ways in the locked cache according to the PLRU rule, and the cache line is marked as 'M'.

If the instruction hits either the normal cache or the locked cache, then the instruction clears all the bytes in the cache line and marks the line as 'M'. The **dcbz_I** instruction has no effect on the L2 cache or the bus.

DCBZ_L Exceptions

The **dcbz_I** instruction causes an alignment exception if the page or the block of the effective address is marked as write-through or cache-inhibited.

The **dcbz_I** instruction is intended to allocate a 32 byte block in the locked cache. When the instruction hits either the normal cache or the locked cache, Espresso sets $HID2[DCHERR] = '1'$. In addition, when the situation happens with $HID2[DCHEE]$, $MSR[EE]$, and $MSR[ME] = '1'$, and $HID2[DCHERR] = '0'$, Espresso also sets $SRR1[10] = '1'$ and raises a machine check.

When $HID2[LCE] = '0'$, the execution of **dcbz_I** causes an illegal instruction exception.

8.2.2.3 *dcbi*

A **dcbi** hit in the locked cache invalidates the cache line and has no effect on the L2 or the bus.

A **dcbi** hit in the normal cache invalidates the cache line. The effect on the L2 and the bus is the same as when $HID2[LCE] = '0'$.

dcbi is a supervisor-level by default and can be set to user-level.

8.2.2.4 *dcbf*

When a **dcbf** hits a modified cache line in either the normal cache or the locked cache, the cache line is castout and is marked 'I'. The effect on the L2 and the bus is the same as when $HID2[LCE] = '0'$.

8.2.2.5 *dcbst*

When a **dcbst** hits a modified cache line in either the locked cache or the normal cache, the cache line is castout and the cache line is marked as 'E'. The effect on the L2 and the bus is the same as when $HID2[LCE] = '0'$.

8.2.2.6 *dcbt and dcbtst*

If a **dcbt** or **dcbtst** hits a cache line in either the normal cache or the locked cache, the instruction is treated as a no-op. If the instruction misses both the locked cache and the normal cache, the corresponding cache line is loaded from the external memory to the normal cache the same as when $HID2[LCE] = '0'$.

8.2.2.7 *Load and Store*

Load and store instructions that miss both the locked and the normal caches result in a cache line load to the normal cache by the PLRU rule among the four ways in the normal cache.

Load and store instructions that hit either the normal cache or the locked cache result in the usual MEI state transition and the PLRU state transition among the four ways in that partition of the cache.

8.3 Direct Memory Access (DMA)

Espresso implements a DMA engine to transfer data between the locked L1 D-Cache and the external memory. The DMA engine has a 15-entry FIFO queue for DMA commands and processes the commands sequentially. The DMA engine operation is controlled by the two special purpose registers: DMAU and DMAL.

The DMA facility is intended to move blocks of data between memory and the L1 D-Cache that are not intended to be shared with other cores and other processes. Espresso does not maintain coherency of this data while it is in the core or being transferred between the core and memory. It is the responsibility of software to ensure that such data is in the locked cache when it is intended to be accessed there, and that such data has been written back to memory before an intended access there. It is a programming error for one core to access data that is in the locked cache of another core, or that is being transferred between memory and the locked cache of another core.

8.3.1 DMA Operation

The DMA engine is disabled at power-on with `HID2[LCE] = '0'`. Setting `HID2[LCE] = '1'` partitions the L1 D-Cache and enables the DMA engine. Note that after `HID2[LCE]` is set to '1', the I-Cache must be invalidated before executing any **dcbz_l** instructions.

When a **mtspr** instruction sets `DMAL[T] = '1'` and `DMAL[F] = '0'`, the DMA engine latches the values in `DMAU` and `DMAL` to form a DMA command, enqueues the command in the DMA queue, and sets `DMAL[T] = '0'`.

`HID2[DMAQL]` indicates the number of DMA commands in the DMA queue, including the command in progress (if any).

When the DMA queue is not empty (that is, `HID2[DMAQL] ≠ '0'`) the DMA engine processes the commands sequentially. The starting address of the transfer in the D-Cache is `DMAL[LC_ADDR] || 0b000000`. The starting address of the transfer in the external memory is `DMAU[MEM_ADDR] || 0b000000`. The number of cache lines to be transferred by a command is `DMAU[DMA_LEN_U] || DMAL[DMA_LEN_L]`, except that a value of zero specifies a length of 128 cache lines. The direction of the transfer is determined by `DMAL[LD]`.

`DMAL[LD] = '0'` means a transfer from the locked cache to the external memory. `DMAL[LD] = '1'` means a transfer from the external memory to the locked cache.

For a DMA store command (that is, `DMAL[LD] = '0'`), the DMA engine performs a D-Cache look-up for each of the cache lines sequentially from the starting address. For a look-up hit in the locked cache, the DMA engine initiates a bus write-with-flush transaction to transfer the 32-byte cache line from the locked cache to the external memory.

For a DMA load command (that is, `DMAL[LD] = '1'`) the DMA engine performs a D-Cache look-up for each of the cache lines sequentially from the starting address. For a look-up hit in the locked cache, the DMA engine initiates a bus burst read transaction to transfer the data from the external memory to the locked cache. For all but the last read transaction associated with the DMA load command, the burst read transaction type is '01011'. The last burst read transaction has a transaction type of '01010'. Espresso initiates the burst transaction type '01011' with the `PFH` signal negated only for the DMA load commands. The memory controller can use the information to pre-fetch the next cache line to improve the performance.

The DMA access to the cache, either a load or a store, results in a PLRU state transition within the 4-way set associated with the cache line, but does not affect the MESI state. If the look-up misses the locked cache, the DMA engine transfers no data and continues to the next cache line.

The **eieio** and **sync** instructions have no effect on the DMA engine.

The only way to flush the DMA queue is to issue a **mtspr** instruction to set `DMAL[F] = '1'`. In this situation, the DMA engine flushes all the commands in the DMA queue, including the command in progress, and sets both `DMAL[F] = DMAL[T] = '0'`. Such an instruction must be followed by a **sync** instruction to ensure that the pending bus transaction associated with the discarded command, if any, complete before the DMA engine accepts the next DMA command.

User-mode access to the DMA facility is supported through an address translation mechanism specific to this purpose. See *Section 2.1.2.4 Direct Memory Access Registers (DMAU and DMAL)* on page 62 for details.

8.3.2 Exception Conditions

There are four conditions under which a DMA operation can cause an exception.

8.3.2.1 DMA Queue Overflow

When a **mtspr** instruction sets $\text{DMAL}[T] = '1'$ and $\text{DMAL}[F] = '0'$ while $\text{HID2}[\text{DMAQL}] = 15$, the DMA engine does not latch the DMA command, but sets $\text{DMAL}[T] = '0'$ and $\text{HID2}[\text{DQOERR}] = '1'$. In addition, when the situation happens that $\text{HID2}[\text{DQOEE}]$, $\text{MSR}[\text{EE}]$, and $\text{MSR}[\text{ME}] = '1'$ and $\text{HID2}[\text{DQOERR}] = '0'$, the Espresso core also sets $\text{SRR1}[10] = '1'$ and raises a machine check.

8.3.2.2 DMA Look-up Hits Normal Cache

When the DMA engine looks up the L1 cache tag and hits in the normal cache partition, Espresso transfers no data, continues to the next cache line, and indicates the situation by setting $\text{HID2}[\text{DNCERR}] = '1'$. In addition, when the situation happens that $\text{HID2}[\text{DNCEE}]$, $\text{MSR}[\text{EE}]$, and $\text{MSR}[\text{ME}] = '1'$ and $\text{HID2}[\text{DNCERR}] = '0'$, the Espresso core also sets $\text{SRR1}[10] = '1'$ and raises a machine check.

8.3.2.3 DMA Look-up Miss

When a DMA engine look-up misses the L1 cache tag, Espresso transfers no data, continues to the next cache line, and indicates the situation by setting $\text{HID2}[\text{DCMERR}] = '1'$. In addition, when the situation happens that $\text{HID2}[\text{DCMEE}]$, $\text{MSR}[\text{EE}]$, and $\text{MSR}[\text{ME}] = '1'$ and $\text{HID2}[\text{DCMERR}] = '0'$, the Espresso core also sets $\text{SRR1}[10] = '1'$ and raises a machine check.

8.3.3 DMA Timing

A DMA command is broken into a sequence of transaction requests. Each request transfers a 32-byte block between the locked cache and the external memory. The read/write transaction requests from the DMA are served by the MIU. On a first-come-first-serve basis, the MIU serves transaction requests from multiple sources, for example, DMA read, instruction load, and so on.

A DMA transaction request requires one cycle to arbitrate for L1 cache tag access and then one cycle to look up the tag for the cache line. After the tag look-up, the DMA makes the transaction request to the MIU.

For a DMA store, it takes one cycle to fetch the 32-byte block from the cache and make the write transaction request to the MIU to transfer the data to the external memory. There is a 2-entry DMA store queue to support the pipelined write transactions.

For a DMA load, there is a 4-entry DMA load queue to support the pipelined read transactions from the external memory. After receiving the 32-byte data from the external memory, it takes one cycle to place the data into the cache.

9. Performance Monitor

The performance monitor facility provides the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache, data cache, or L2 cache, types of instructions dispatched, mispredicted branches, and other occurrences. The count of such events (which might be an approximation) can be used to trigger the performance monitor exception. The performance monitor facility is not defined by the PowerPC Architecture.

The performance monitor can be used for the following:

- To increase system performance with efficient software, especially in a multiprocessing system. Memory hierarchy behavior can be monitored and studied to develop algorithms that schedule tasks (and perhaps partition them) and that structure and distribute data optimally.
- To improve processor architecture, the detailed behavior of the Espresso structure must be known and understood in many software environments. Some environments cannot be easily characterized by a benchmark or trace.
- To help system developers bring up and debug their systems.

The performance monitor uses the following the Espresso-specific special-purpose registers (SPRs):

- The performance monitor counter registers (PMC1 - PMC4) are used to record the number of times a certain event has occurred. UPMC1 - UPMC4 provide user-level read access to these registers.
- The monitor mode control registers (MMCR0, MMCR1) are used to enable various performance monitor interrupt functions and select events to count. UMMCR0 and UMMCR1 provide user-level read access to these registers.
- The sampled instruction address register (SIA) contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. USIA provides user-level read access to the SIA.

Four 32-bit counters in each Espresso core count occurrences of software-selectable events. Two control registers (MMCR0 and MMCR1) are used to control performance monitor operation. The counters and the control registers are supervisor-level SPRs; however, in Espresso, the contents of these registers can be read by user-level software using separate SPRs (UMMCR0 and UMMCR1). Control fields in the MMCR0 and MMCR1 select the events to be counted, can enable a counter overflow to initiate a performance monitor exception, and specify the conditions under which counting is enabled.

As with other PowerPC exceptions, the performance monitor interrupt follows the normal PowerPC exception model with a defined exception vector offset (x'00F00'). Its priority is below the external interrupt and above the decremter interrupt.

9.1 Performance Monitor Interrupt

The performance monitor provides the ability to generate a performance monitor interrupt triggered by a counter overflow condition in one of the performance monitor counter registers (PMC1 - PMC4) (see *Section 9.2.5* on page 247). A counter is considered to have overflowed when its most-significant bit is set. A performance monitor interrupt can also be caused by the flipping from 0 to 1 of certain bits in the time base register, which provides a way to generate a time reference-based interrupt.

Although the interrupt signal condition can occur with MSR[EE] = '0', the actual exception cannot be taken until MSR[EE] = '1'.

When a performance monitor exception is taken, the action taken depends on the programmable events, as follows. To help track which part of the code was being executed when an exception was signaled, the address of the last completed instruction during that cycle is saved in the SIA. The SIA is not updated if no instruction completed the cycle in which the exception was taken.

Exception handling for the performance monitor interrupt exception is described in *Section 4.5.12 Performance Monitor Interrupt (x'00F00')* on page 149.

9.2 Special-Purpose Registers Used by Performance Monitor

The performance monitor incorporates the SPRs listed in *Table 9-1*. All of these supervisor-level registers are accessed through **mtspr** and **mfspr** instructions.

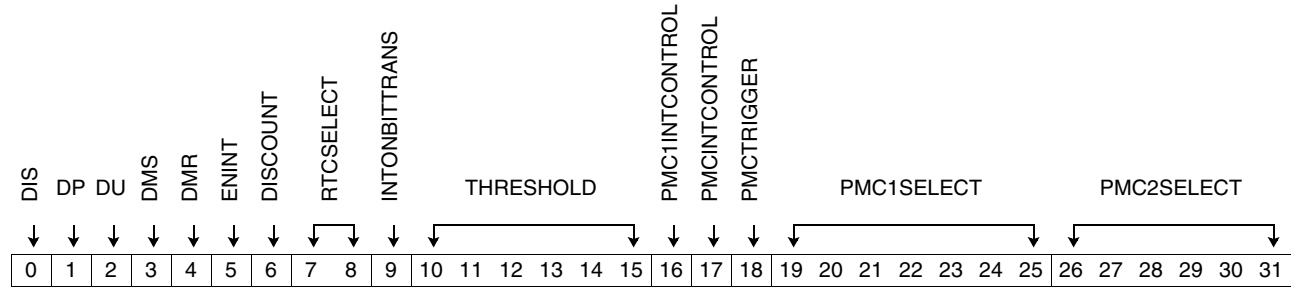
Table 9-1. Performance Monitor SPRs

SPR Number	spr[5-9] spr[0-4]	Register Name	Access Level
952	'11101 11000'	MMCR0	Supervisor
953	'11101 11001'	PMC1	Supervisor
954	'11101 11010'	PMC2	Supervisor
955	'11101 11011'	SIA	Supervisor
956	'11101 11100'	MMCR1	Supervisor
957	'11101 11101'	PMC3	Supervisor
958	'11101 11110'	PMC4	Supervisor
936	'11101 01000'	UMMCR0	User (read only)
937	'11101 01001'	UPMC1	User (read only)
938	'11101 01010'	UPMC2	User (read only)
939	'11101 01011'	USIA	User (read only)
940	'11101 01100'	UMMCR1	User (read only)
941	'11101 01101'	UPMC3	User (read only)
942	'11101 01110'	UPMC4	User (read only)

The following sections describe the registers used by the performance monitor.

9.2.1 Monitor Mode Control Register 0 (MMCR0)

The Monitor Mode Control Register 0 (MMCR0) is a 32-bit SPR provided to specify events to be counted and recorded. MMCR0 can be written to only in supervisor mode. User-level software can read the contents of MMCR0 by issuing an **mf spr** instruction to UMMCR0, described in *Section 9.2.2* on page 246. This register must be cleared at power up. Reading this register does not change its contents.



Bits	Field Name	Description
0	DIS	Disables counting unconditionally. 0 The values of the PMC n counters can be changed by hardware. 1 The values of the PMC n counters cannot be changed by hardware.
1	DP	Disables counting while in supervisor mode. 0 The PMC n counters can be changed by hardware. 1 If the processor is in supervisor mode (MSR[PR] is cleared), the counters are not changed by hardware.
2	DU	Disables counting while in user mode. 0 The PMC n counters can be changed by hardware. 1 If the processor is in user mode (MSR[PR] is set), the PMC n counters are not changed by hardware.
3	DMS	Disables counting while MSR[PM] is set. 0 The PMC n counters can be changed by hardware. 1 If MSR[PM] is set, the PMC n counters are not changed by hardware.
4	DMR	Disables counting while MSR[PM] is zero. 0 The PMC n counters can be changed by hardware. 1 If MSR[PM] is cleared, the PMC n counters are not changed by hardware.
5	ENINT	Enables performance monitor interrupt signaling. 0 Interrupt signaling is disabled. 1 Interrupt signaling is enabled. Cleared by hardware when a performance monitor interrupt is taken. To re-enable these interrupt signals, software must set this bit after servicing the performance monitor interrupt. The IPL ROM code clears this bit before passing control to the operating system.
6	DISCOUNT	Disables counting of PMC n when a performance monitor interrupt is signaled (that is, ((PMC n INTCONTROL = 1) & (PMC n [0] = 1) & (ENINT = 1)) or the occurrence of an enabled time base transition with ((INTONBITTRANS = 1) & (ENINT = 1)). 0 Signaling a performance monitor interrupt does not affect the counting status of PMC n . 1 The signaling of a performance monitor interrupt prevents the changing of the PMC1 counter. The PMC n counter does not change if PMC2COUNTCTL = '0'. Because a time base signal could have occurred along with an enabled counter overflow condition, software should always reset INTONBITTRANS to zero, if the value in INTONBITTRANS was a one.

Bits	Field Name	Description
7:8	RTCSELECT	Time base lower (TBL) bit selection enable 00 Pick bit 31 to count. 01 Pick bit 23 to count. 10 Pick bit 19 to count. 11 Pick bit 15 to count.
9	INTONBIT-TRANS	Causes interrupt signaling on bit transition (identified in RTCSELECT) from off to on. 0 Do not allow an interrupt signal on the transition of a chosen bit. 1 Signal an interrupt on the transition of a chosen bit. Software is responsible for setting and clearing INTONBITTRANS.
10:15	THRESHOLD	Threshold value. All 6 bits are supported by Espresso enabling threshold values from 0 to 63. The intent of the THRESHOLD support is to characterize L1 data cache misses.
16	PMC1INT-CONTROL	Enables interrupt signaling due to PMC1 counter overflow. 0 Disable PMC1 interrupt signaling due to PMC1 counter overflow. 1 Enable PMC1 Interrupt signaling due to PMC1 counter overflow.
17	PMCINT-CONTROL	Enable interrupt signaling due to any PMC2 - PMC4 counter overflow. Overrides the setting of DISCOUNT. 0 Disable PMC2 - PMC4 interrupt signaling due to PMC2 - PMC4 counter overflow. 1 Enable PMC2 - PMC4 interrupt signaling due to PMC2 - PMC4 counter overflow.
18	PMC-TRIGGER	Can be used to trigger counting of PMC2 - PMC4 after PMC1 has overflowed or after a performance monitor interrupt is signaled. 0 Enable PMC2 - PMC4 counting. 1 Disable PMC2 - PMC4 counting until either PMC1[0] = '1' or a performance monitor interrupt is signaled.
19:25	PMC1-SELECT	PMC1 input selector, 128 events selectable. See <i>Table 9-2</i> on page 248.
26:31	PMC2-SELECT	PMC2 input selector, 64 events selectable. See <i>Table 9-3</i> on page 249.

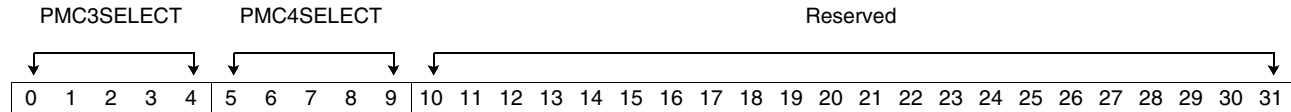
MMCR0 can be accessed with the **mtspr** and **mfspir** instructions using SPR 952.

9.2.2 User Monitor Mode Control Register 0 (UMMCR0)

The contents of MMCR0 are reflected to UMMCR0, which can be read by user-level software. UMMCR0 can be accessed with the **mfspir** instructions using SPR 936.

9.2.3 Monitor Mode Control Register 1 (MMCR1)

The Monitor Mode Control Register 1 (MMCR1) functions as an event selector for Performance Monitor Counter Registers 3 and 4 (PMC3 and PMC4).



Bits	Field Name	Description
0:4	PMC3 SELECT	PMC3 input selector. 32 events selectable. See <i>Table 9-4</i> for defined selections.
5:9	PMC4 SELECT	PMC4 input selector. 32 events selectable. See <i>Table 9-5</i> for defined selections.
10:31	—	Reserved

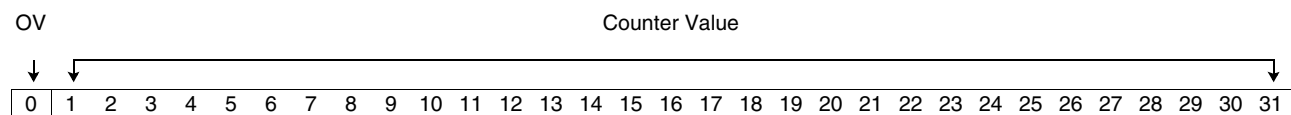
MMCR1 can be accessed with the **mtspr** and **mfspir** instructions using SPR 956. User-level software can read the contents of MMCR1 by issuing an **mfspir** instruction to UMMCR1, described in *Section 9.2.4*.

9.2.4 User Monitor Mode Control Register 1 (UMMCR1)

The contents of MMCR1 are reflected to UMMCR1, which can be read by user-level software. UMMCR1 can be accessed with the **mfspir** instructions using SPR 940.

9.2.5 Performance Monitor Counter Registers (PMC1 - PMC4)

PMC1 - PMC4 are 32-bit counters that can be programmed to generate interrupt signals when they overflow.



Bits	Field Name	Description
0	OV	Overflow. When this bit is set, it indicates this counter has reached its maximum value.
1:31	Counter Value	Indicates the number of occurrences of the specified event.

Counters overflow when the high-order bit (the sign bit) becomes set; that is, they reach the value of x'8000_0000' (2,147,483,648 in decimal). However, an interrupt is not signaled unless both MMCR0[ENINT] and either PMC1INTCONTROL or PMCINTCONTROL in the MMCR0 register are also set as appropriate.

Note: The interrupts can be masked by clearing MSR[EE]; the interrupt signal condition can occur with MSR[EE] cleared, but the exception is not taken until MSR[EE] is set. Setting MMCR0[DISCOUNT] forces counters to stop counting when a counter interrupt occurs.

Software is expected to use the **mtspr** instruction to explicitly set PMC to nonoverflowed values. Setting an overflowed value can cause an erroneous exception. For example, if both MMCR0[ENINT] and either PMC1INTCONTROL or PMCINTCONTROL are set and the **mtspr** instruction loads an overflow value, an interrupt signal can be generated without event counting having taken place.

The events to be monitored can be chosen by setting MMCR0[19:31] and MMCR1[0:9]. The selected events are counted beginning when MMCR0 is set until either MMCR0 is reset or a performance monitor interrupt is generated.

Note: Each event whose encoding is less than 20 is specific to a single core; each event whose encoding is 20 or greater corresponds to activity associated with all cores. To avoid an initial spurious count when programming any event associated with all cores (events numbered 20 or greater), disable counting while configuring the control registers. Then enable counting after the control registers have been configured. Counting is disabled by setting MMCR0[DIS] = '1'.

Bits MMCR0[19:25] specify events associated with PMC1, as shown in *Table 9-2*.

Table 9-2. PMC1 Events—MMCR0[19:25] Select Encodings

Encoding	Description
0000000	Register holds current value.
0000001	Number of processor cycles.
0000010	Number of instructions that have completed. Does not include folded branches.
0000011	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTC-SELECT, MMCR0[7:8]. 00 31 01 23 10 19 11 15
0000100	Number of instructions dispatched: 0, 1, or 2 instructions per cycle.
0000101	Number of eieio instructions completed.
0000110	Number of cycles spent performing table search operations for the ITLB.
0000111	Number of accesses that hit the L2. This event includes cache ops (that is, dcbz).
0001000	Number of valid instruction EAs delivered to the memory subsystem.
0001001	Number of times the address of an instruction being completed matches the address in the IABR.
0001010	Number of loads that miss the L1 with latencies that exceeded the threshold value.
0001011	Number of branches that are unresolved when processed.
0001100	Number of cycles the dispatcher stalls due to a second unresolved branch in the instruction stream.
0001110	Number of times a store to a shared line occurs in the L1 cache.
0001111	Number of times the L2 cache supplies shared intervention data.
0010000	Number of times a paradox between L1 and L2 is detected.
0010001 - 0010011	Reserved.
0010100	All cores: number of load requests to the CIU.
0010101	All cores: number of address only requests to the BIU.

Table 9-2. PMC1 Events—MMCR0[19:25] Select Encodings

Encoding	Description
0010110	All cores: number of times a paradox is detected in the CIU.
0010111	All cores: number of data beats on the 60Xe bus.
All others	Reserved. Might be used in a later revision.

Bits MMCR0[26:31] specify events associated with PMC2, as shown in Table 9-3.

Table 9-3. PMC2 Events—MMCR0[26:31] Select Encodings

Encoding	Description
000000	Register holds current value.
000001	Number of processor cycles.
000010	Number of instructions that have completed. Does not include folded branches.
000011	Time-base (lower) bit transitions. Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTC-SELECT, MMCR0[7:8]. 00 31 01 23 10 19 11 15
000100	Number of instructions dispatched: 0, 1, or 2 instructions per cycle.
000101	Number of L1 I-Cache misses. Indicates the number of times an instruction fetch missed the L1 instruction cache.
000110	Number of ITLB misses. Indicates the number of times the needed instruction address translation was not in the ITLB.
000111	L2 I-misses. Counts the number of accesses which miss the L2 due to an I-side request.
001000	Number of fall-through branches. Indicates the number of branches that were predicted not taken.
001001	Reserved.
001010	Reserved loads. Incremented every time that a reserved load completes.
001011	Loads and stores. Counts all load and store instructions completed.
001100	Number of snoops. Gives the total number of snoops to the L1 and the L2.
001101	L1 castouts to L2. Number of times the L1 castout goes to the L2.
001110	System unit instructions. Number of system unit instructions completed.
001111	Instruction miss cycles. Counts the total number of L1 miss cycles of instruction fetches.
010000	First speculative branch resolved correctly. Indicates the number of branches that allow speculative execution beyond those that resolved correctly
010001	Number of times a store to a shared line occurs in the L2.
010010	Number of times the L1 supplies shared intervention data.
010011	Reserved.
010100	All cores: number of store requests to the CIU.
010101	All cores: number of cycles in which the number of outstanding BIU transactions exceeds a threshold.
010110	All cores: number of times a modified intervention is processed in the CIU.
All others	Reserved.

Bits MMCR1[0:4] specify events associated with PMC3, as shown in *Table 9-4*.

Table 9-4. PMC3 Events—MMCR1[0:4] Select Encodings

Encoding	Description
00000	Register holds current value.
00001	Number of processor cycles.
00010	Number of completed instructions, not including folded branches.
00011	Number of transitions from 0 to 1 of specified bits in the time base lower (TBL) register. Bits are specified through RTC-SELECT, MMCR0[7:8]. 00 31 01 23 10 19 11 15
00100	Number of instructions dispatched. 0, 1, or 2 per cycle.
00101	Number of L1 data cache misses. Does not include cache operations.
00110	Number of DTLB misses.
00111	Number of L2 data misses.
01000	Number of predicted branches that were taken.
01001	Reserved.
01010	Number of store conditional instructions completed.
01011	Number of instructions completed from the FPU.
01100	Number of L2 castouts caused by snoops to modified lines.
01101	Number of cache operations that hit in the L2 cache.
01110	Reserved.
01111	Number of cycles generated by L1 load misses.
10000	Number of branches in the second speculative stream that resolve correctly.
10001	Number of cycles the BPU stalls due to LR or CR unresolved dependencies.
10010	Number of times the L1 supplies modified intervention data.
10011	Number of icbi snoops.
10100	All cores: number of address-only requests to the CIU.
10101	All cores: number of load requests to the BIU.
10110	All cores: number of times a shared intervention is processed in the CIU.
All others	Reserved. Might be used in a later revision.

Bits MMCR1[5:9] specify events associated with PMC4, as shown in *Table 9-5*.

Table 9-5. PMC4 Events—MMCR1[5:9] Select Encodings

Encoding	Comments
00000	Register holds current value.
00001	Number of processor cycles.
00010	Number of completed instructions, not including folded branches.
00011	Number of transitions from 0 to 1 of specified bits in the Time Base Lower (TBL) Register. Bits are specified through RTCSELECT and MMCR0[7:8]. 00 31 01 23 10 19 11 15
00100	Number of instructions dispatched: 0, 1, or 2 per cycle.
00101	Number of L2 castouts.
00110	Number of cycles spent performing table searches for DTLB accesses.
00111	Reserved. Might be used in a later revision.
01000	Number of mispredicted branches.
01001	Reserved. Might be used in a later revision.
01010	Number of store conditional instructions completed with reservation intact.
01011	Number of completed sync instructions.
01100	Number of snoop request retries.
01101	Number of completed integer operations.
01110	Number of cycles the BPU cannot process new branches due to having two unresolved branches.
01111	Reserved
10000	Number of times the L2 supplies modified intervention data.
10001	Number of tlbie snoops.
10010	Number of times the L2 bank refresh counter overflows.
10011	Reserved.
10100	All cores: number of ARTRYs that occur in the CIU.
10101	All cores: number of store requests to the BIU.
10110	All cores: number of times a shared intervention from two cores is processed in the CIU.
All others	Reserved. Might be used in a later revision.

The PMC registers can be accessed with the **mtspr** and **mfspir** instructions using the following SPR numbers:

- PMC1 is SPR 953
- PMC2 is SPR 954
- PMC3 is SPR 957
- PMC4 is SPR 958

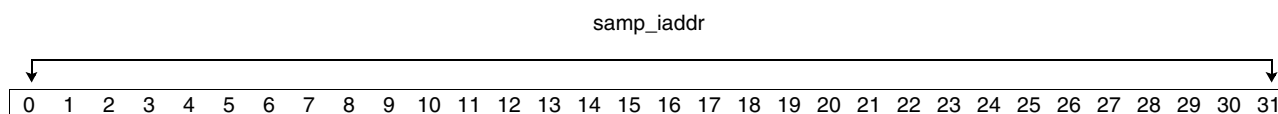
9.2.6 User Performance Monitor Counter Registers (UPMC1 - UPMC4)

The contents of the PMC1 - PMC4 are reflected to UPMC1 - UPMC4, which can be read by user-level software. The UPMC registers can be read with the **mfspir** instructions using the following SPR numbers:

- UPMC1 is SPR 937
- UPMC2 is SPR 938
- UPMC3 is SPR 941
- UPMC4 is SPR 942

9.2.7 Sampled Instruction Address Register (SIA)

The Sampled Instruction Address Register (SIA) is a supervisor-level register that contains the effective address of an instruction executing at or around the time that the processor signals the performance monitor interrupt condition. The SIA is shown as follows:



Bits	Field Name	Description
0:31	samp_iaddr	Sampled instruction address

If the performance monitor interrupt is triggered by a threshold event, the SIA contains the address of the exact instruction (called the sampled instruction) that caused the counter to overflow.

If the performance monitor interrupt was caused by something besides a threshold event, the SIA contains the address of the last instruction completed during that cycle. SIA can be accessed with the **mtspr** and **mfspir** instructions using SPR 955.

9.2.8 User Sampled Instruction Address Register (USIA)

The contents of SIA are reflected to USIA, which can be read by user-level software. USIA can be accessed with the **mfspir** instructions using SPR 939.

9.3 Event Counting

Counting can be enabled if conditions in the processor state match a software-specified condition. Because a software task scheduler can switch a processor's execution among multiple processes and because statistics on only a particular process might be of interest, a facility is provided to mark a process. The performance monitor (PM) bit, MSR[29] is used for this purpose. System software can set this bit when a marked process is running. This enables statistics to be gathered only during the execution of the marked process. The states of MSR[PR] and MSR[PM] together define a state that the processor (supervisor or program) and the process (marked or unmarked) can be in at any time. If this state matches a state specified by the MMCR, the state for which monitoring is enabled, counting is enabled.

The following are states that can be monitored:

- (Supervisor) only
- (User) only
- (Marked and user) only
- (Not marked and user) only
- (Marked and supervisor) only
- (Not marked and supervisor) only
- (Marked) only
- (Not marked) only

In addition, one of two unconditional counting modes can be specified:

- Counting is unconditionally enabled regardless of the states of MSR[PM] and MSR[PR]. This can be accomplished by clearing MMCR0[0:4].
- Counting is unconditionally disabled regardless of the states of MSR[PM] and MSR[PR]. This is done by setting MMCR0[0].

The performance monitor counters count specified events and are used to generate performance monitor exceptions when an overflow (most-significant bit is a '1') situation occurs. The Espresso performance monitor has four, 32-bit registers that can count up to x'7FFFFFFF' (2,147,483,648 in decimal) before overflowing. Bit 0 of the register is used to determine when an interrupt condition exists.

9.4 Event Selection

Event selection is handled through MMCR0 and MMCR1, described in *Section 9.2.1 Monitor Mode Control Register 0 (MMCR0)* on page 245 and *Section 9.2.3 Monitor Mode Control Register 1 (MMCR1)* on page 247. Event selection is described as follows:

- The four event-select fields in MMCR0 and MMCR1 are as follows:
 - MMCR0[19:25] (PMC1SELECT). PMC1 input selector, 128 events selectable; not all are defined. See *Table 9-2* on page 248.
 - MMCR0[26:31] (PMC2SELECT). PMC2 input selector, 64 events selectable; not all are defined. See *Table 9-3* on page 249.
 - MMCR0[0:4] (PMC3SELECT). PMC3 input selector. 32 events selectable; not all are defined. See *Table 9-4* on page 250.
 - MMCR0[5:9] (PMC4SELECT). PMC4 input selector. 32 events selectable; not all are defined. See *Table 9-5* on page 251.

- In *Table 9-2* on page 248, *Table 9-3* on page 249, *Table 9-4* on page 250, and *Table 9-5* on page 251, a correlation is established between each counter, events to be traced, and the pattern required for the desired selection.
- The first five events are common to all four counters and are considered to be reference events. These are as follows:
 - 00000 Register holds current value
 - 00001 Number of processor cycles
 - 00010 Number of completed instructions, not including folded branches
 - 00011 Number of transitions from 0 to 1 of specified bits in the Time Base Lower (TBL) Register. Bits are specified through RTCSELECT and MMCR0[7:8].
 - 00 = 31
 - 01 = 23
 - 10 = 19
 - 11 = 15
 - 00100 Number of instructions dispatched. 0, 1, or 2 per cycle
- Some events can have multiple occurrences per cycle, and therefore need 2 or 3 bits to represent them.

9.5 Notes

The following warnings should be noted:

- Only those load and store instructions in queue position 0 of their corresponding load/store queues are monitored when a threshold event is selected in PMC1.
- The Espresso microprocessor cannot accurately track threshold events with respect to the following types of loads and stores:
 - Unaligned load and store operations that cross a word boundary
 - Load and store multiple operations
 - Load and store string operations

10. Espresso PowerPC Instruction Set

This section lists the PowerPC instruction set in alphabetical order by mnemonic. Each entry includes:

- Instruction format. The format diagrams show all valid combinations of instruction fields; to provide a graphical representation of the instruction formats.
- Quick reference legend that provides the level of the PowerPC Architecture in which the instruction can be found UISA (user instruction set architecture), VEA (virtual environment architecture), and OEA (operating environment architecture)
- Privilege level of the instruction: user-level or supervisor-level. Note that an instruction is assumed to be user-level unless the legend specifies that it is supervisor-level.

A description of the instruction fields and pseudocode conventions are also provided.

Note: The architecture specification refers to user-level as problem state and supervisor-level as privileged state.

10.1 Instruction Formats

Instructions are 4 bytes long and word-aligned, so that when instruction addresses are presented to the processor (as in branch instructions) the 2 low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its 2 low-order bits are zero.

Bits 0:5 always specify the primary opcode. Many instructions also have an extended opcode. The remaining bits of the instruction contain one or more fields for the different instruction formats.

Some instruction fields are reserved, or must contain a predefined value as shown in the individual instruction layouts. If a reserved field does not have all bits cleared, or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are described in the addressing modes and instruction set summary section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Within the instruction format diagram, the instruction operation code and extended operation code (if extended form) are specified in decimal. These fields have been converted to hexadecimal and are shown on line two for each instruction definition.

10.1.1 Split-Field Notation

Some instruction fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Such a field is called a split field. Split fields that represent the concatenation of the sequences from left to right are shown in lowercase letters. These split fields (spr and tbr) are described in *Table 10-1*.

Table 10-1. Split-Field Notation and Conventions

Field	Description
spr (11–20)	This field specifies a special-purpose register for the mtspr and mfspir instructions. The encoding is described in the Move To/From Special-Purpose Register instructions (OEA) section of the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.
tbr (11–20)	This field specifies either the time base lower (TBL) or time base upper (TBU).

10.1.2 Instruction Fields

Table 10-2 describes the instruction fields used in the various instruction formats.

Table 10-2. Instruction Syntax Conventions (Page 1 of 2)

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address (CIA). For more information about the CIA, see <i>Table 10-3</i> on page 257. The effective (logical) address of the branch is either the sum of the LI field sign-extended to 32 bits and the address of the branch instruction or the sum of the BD field sign-extended to 32 bits and the address of the branch instruction.
	1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 32 bits or the BD field sign-extended to 32 bits. Note: The LI and BD fields are sign-extended to 32 bits.
BD (16:29)	Immediate field that specifies a 14-bit signed twos complement branch displacement that is concatenated on the right with '00' and sign-extended to 32 bits.
BI (11:15)	This field specifies a bit in the CR to be used as the condition of a branch conditional instruction.
BO (6:10)	This field specifies options for the branch conditional instructions. The encoding is described in the conditional branch control section of the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.
crbA (11:15)	This field specifies a bit in the CR to be used as a source.
crbB (16:20)	This field specifies a bit in the CR to be used as a source.
crbD (6:10)	This field specifies a bit in the CR, or in the FPSCR, as the destination of the result of an instruction.
crfD (6:8)	This field specifies one of the CR fields, or one of the FPSCR fields, as a destination.
crfS (11:13)	This field specifies one of the CR fields, or one of the FPSCR fields, as a source.
CRM (12:19)	This field mask identifies the CR fields that are to be updated by the mtcrf instruction.
d (16:31 or 20:31)	Immediate field that specifies a signed twos complement integer that is sign-extended to 32 bits.
FM (7:14)	This field mask identifies the FPSCR fields that are to be updated by the mtfsf instruction.
frA[11:15]	This field specifies an FPR as a source.
frB[16:20]	This field specifies an FPR as a source.
frC[21:25]	This field specifies an FPR as a source.
frD[6:10]	This field specifies an FPR as the destination.
frS[6:10]	This field specifies an FPR as a source.
I (17:19, or 22:24)	This field specifies a GQR control register that is used by the paired single load or store instructions.
IMM (16:19)	Immediate field used as the data to be placed into a field in the FPSCR.
LI (6:29)	Immediate field that specifies a 24-bit signed twos complement integer that is concatenated on the right with '00' and sign-extended to 32 bits.
LK (31)	Link bit.
	0 Does not update the link register (LR). 1 Updates the LR. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the LR.
MB (21:25) and ME (26:30)	These fields are used in rotate instructions to specify a 32-bit mask in the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.
NB (16:20)	This field specifies the number of bytes to move in an immediate string load or store.
OE (21)	This field is used for extended arithmetic to enable setting OV and SO in the XER.
OPCD (0:5)	Primary opcode field.

Table 10-2. Instruction Syntax Conventions (Page 2 of 2)

Field	Description
rA[11:15]	This field specifies a GPR to be used as a source or destination.
rB[16:20]	This field specifies a GPR to be used as a source.
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 Updates the CR to reflect the result of the operation. For integer instructions, CR bits 0:2 are set to reflect the result as a signed quantity and CR bit 3 receives a copy of the summary overflow bit, XER[SO]. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR bits 4:7 are set to reflect floating-point exception, floating-point enabled exception, floating-point invalid operation exception, and floating-point overflow exception. Note: Exceptions are referred to as interrupts in the architecture specification.
rD[6:10]	This field specifies a GPR to be used as a destination.
rS[6:10]	This field specifies a GPR to be used as a source.
SH (16:20)	This field specifies a shift amount.
SIMM (16:31)	This immediate field specifies a 16-bit signed integer.
SR (12:15)	This field specifies one of the 16 segment registers.
TO (6:10)	This field specifies the conditions on which to trap. The encoding is described in the trap instructions section in the <i>PowerPC Microprocessor Family: The Programming Environments</i> manual.
UIMM (16:31)	This immediate field specifies a 16-bit unsigned integer.
XO (21:30, 22:30, 25:30, or 26:30)	Extended opcode field.

10.1.3 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). Table 10-3 provides a list of pseudocode notation and conventions used throughout this section.

Table 10-3. Notation and Conventions (Page 1 of 3)

Notation/Convention	Meaning
$\leftarrow$	Assignment.
$\leftarrow \text{iea}$	Assignment of a 32-bit instruction effective address.
$\neg$	NOT logical operator.
$*, \times$	Multiplication.
$\div$	Division (yielding quotient).
$+$	Twos-complement addition.
$-$	Twos-complement subtraction, unary minus.
$=, \neq$	Equals and not equals relations.
$<, \leq, \geq, >$	Signed comparison relations.
$\cdot$ (period)	Update. When used as a character of an instruction mnemonic, a period ($\cdot$) means that the instruction updates the condition register field.
c	Carry. When used as a character of an instruction mnemonic, a "c" indicates a carry out in XER[CA].

Table 10-3. Notation and Conventions (Page 2 of 3)

Notation/Convention	Meaning
e	Extended precision. When used as the last character of an instruction mnemonic, an "e" indicates the use of XER[CA] as an operand in the instruction and records a carry out in XER[CA].
o	Overflow. When used as a character of an instruction mnemonic, an "o" indicates the record of an overflow in XER[OV] and CR0[SO] for integer instructions or CR1[SO] for floating-point instructions.
<U, >U	Unsigned comparison relations.
?	Unordered comparison relation.
&,	AND, OR logical operators.
	Used to describe the concatenation of two values (that is, 010 111 is the same as '010111').
$\oplus$, $\equiv$	Exclusive-OR, Equivalence logical operators (for example, $(a \equiv b) = (a \oplus \neg b)$).
0bnnnn	A number expressed in binary format.
0xnnnn or x'nnnn nnnn'	A number expressed in hexadecimal format.
(n)x	The replication of x, n times (that is, x concatenated to itself n - 1 times). (n)0 and (n)1 are special cases. A description of the special cases follows: • (n)0 means a field of n bits with each bit equal to '0'. Thus (5)0 is equivalent to '00000'. • (n)1 means a field of n bits with each bit equal to '1'. Thus (5)1 is equivalent to '11111'.
(rAl0)	The contents of rA if the rA field has the value 1 - 31, or the value '0' if the rA field is '0'.
(rX)	The contents of rX.
x[n]	n is a bit or field within x, where x is a register.
x^n	x is raised to the nth power.
ABS(x)	Absolute value of x.
CEIL(x)	Least integer $\leq$ x.
Characterization	Reference to the setting of status bits in a standard way that is explained in the text.
CIA	Current instruction address. The 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK = '1' to set the link register. Does not correspond to any architected register.
Clear	Clear the leftmost or rightmost n bits of a register to 0. This operation is used for rotate and shift instructions.
Clear left and shift left	Clear the leftmost b bits of a register, then shift the register left by n bits. This operation can be used to scale a known nonnegative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits are set to 0.
Do	Do loop. • Indenting shows range. • "To" and/or "by" clauses specify incrementing an iteration variable. • "While" clauses give termination conditions.
DOUBLE(x)	Result of converting x from floating-point, single-precision format to floating-point, double-precision format.
Extract	Select a field of n bits starting at bit position b in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.
EXTS(x)	Result of extending x on the left with sign bits.
GPR(x)	General-purpose register x.

Table 10-3. Notation and Conventions (Page 3 of 3)

Notation/Convention	Meaning
if...then...else...	Conditional execution, indenting shows range, else is optional.
Insert	Select a field of n bits in the source register, insert this field starting at bit position b of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on doublewords; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. Note: Simplified mnemonics are referred to as extended mnemonics in the architecture specification.
Leave	Leave innermost do loop, or the do loop described in leave statement.
MASK(x, y)	Mask having ones in positions x through y (wrapping if $x > y$) and zeros elsewhere.
MEM(x, y)	Contents of y bytes of memory starting at address x .
NIA	Next instruction address. This is the 32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to the NIA. For instructions that do not branch, the next instruction address is CIA + 4. NIA does not correspond to any architected register.
OEA	PowerPC operating environment architecture.
Rotate	Rotate the contents of a register right or left n bits without masking. This operation is used for rotate and shift instructions.
ROTL(x, y)	Result of rotating the value x left y positions, where x is 32 bits long.
Set	Bits are set to 1.
Shift	Shift the contents of a register right or left n bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.
SINGLE(x)	Result of converting x from floating-point, double-precision format to floating-point, single-precision format.
SPR(x)	Special-purpose register x .
TRAP	Invoke the system trap handler.
Undefined	An undefined value. The value can vary from one implementation to another, and from one execution to another on the same implementation.
UISA	PowerPC user instruction set architecture.
VEA	PowerPC virtual environment architecture.

Table 10-4 describes instruction field notation conventions used throughout this section.

Table 10-4. Instruction Field Conventions (Page 1 of 2)

The Architecture Specification	Equivalent to:
BA, BB, BT	crbA, crbB, crbD
BF, BFA	crfD, crfS
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	frA, frB, frC, frD, frS
FXM	CRM
RA, RB, RT, RS	rA, rB, rD, rS
SI	SIMM

Table 10-4. Instruction Field Conventions (Page 2 of 2)

The Architecture Specification	Equivalent to:
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Precedence rules for pseudocode operators are summarized in Table 10-5.

Table 10-5. Precedence Rules

Operators	Associativity
$x[n]$, function evaluation	Left to right
$(n)x$ or replication, $x(n)$ or exponentiation	Right to left
unary $-$, $\neg$	Right to left
$\times$, $\div$	Left to right
$+$, $-$	Left to right
$ $	Left to right
$=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $<U$, $>U$, $?$	Left to right
$\&$, $\oplus$, $\equiv$	Left to right
$ $	Left to right
$-$ (range)	None
$\leftarrow$, $\leftarrow_{iea}$	None

Operators higher in Table 10-5 are applied before those lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. For example, “ $-$ ” (unary minus) associates from left to right, so $a - b - c = (a - b) - c$. Parentheses are used to override the evaluation order implied by Table 10-5, or to increase clarity; parenthesized expressions are evaluated before serving as operands. Note that the all pseudocode examples provided in this section are for 32-bit implementations of the PowerPC instruction set

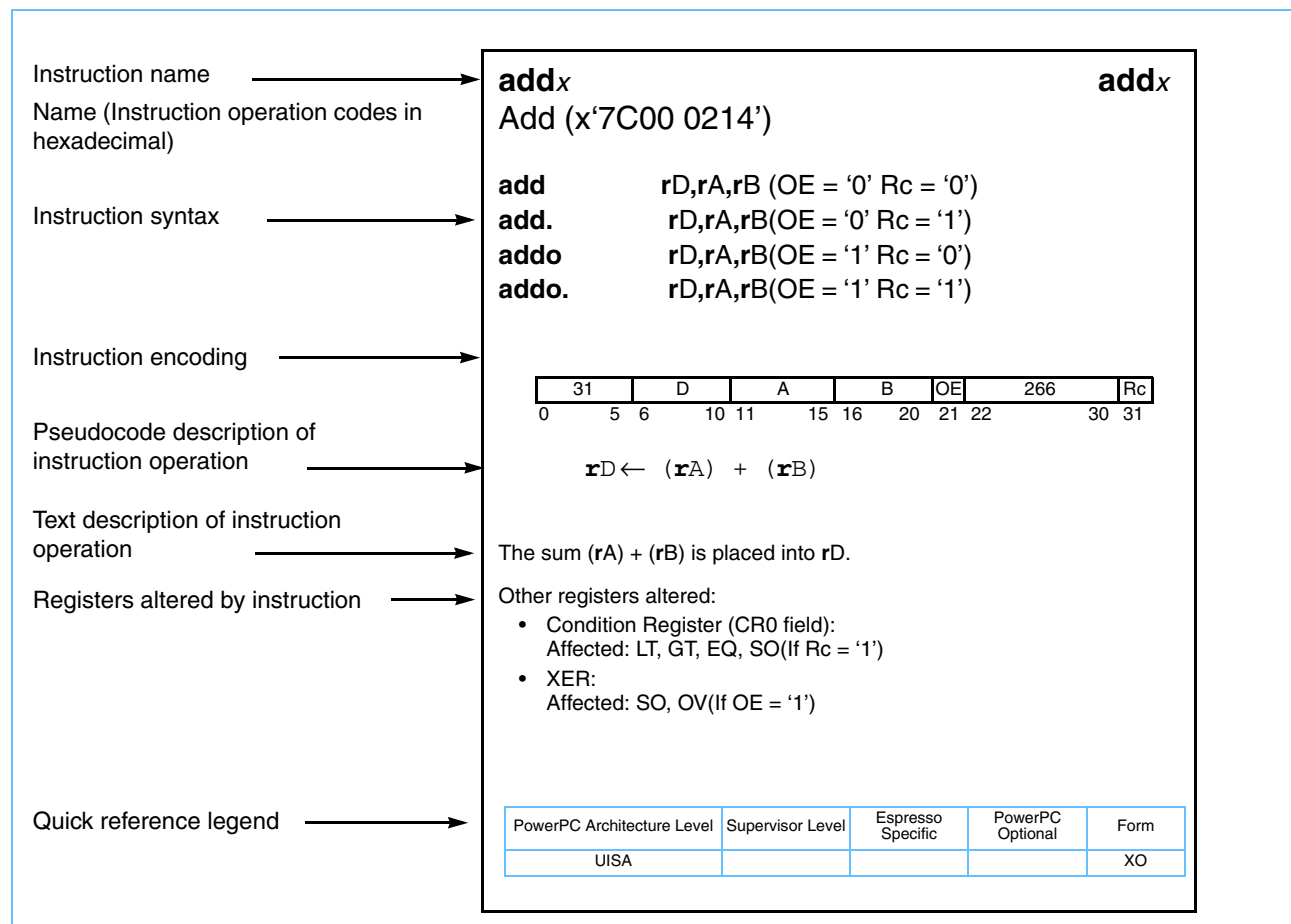
10.1.4 Computation Modes

The PowerPC Architecture is defined for 32-bit implementations, in which all registers except the FPRs are 32 bits long, and effective addresses are 32 bits long. The FPR registers are 64 bits long. For more information about computation modes see the computation modes section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

10.2 PowerPC Instruction Set

The remainder of this section lists and describes the instruction set for the PowerPC Architecture. The instructions are listed in alphabetical order by mnemonic. *Figure 10-1* shows the format for each instruction description page.

Figure 10-1. Instruction Description



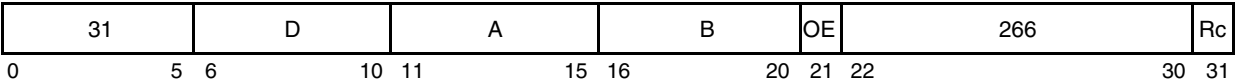
Note: The execution unit that executes the instruction might not be the same for all PowerPC processors.

add_x

Add (x'7C00 0214')

add_x

add	rD,rA,rB	(OE = '0' Rc = '0')
add.	rD,rA,rB	(OE = '0' Rc = '1')
addo	rD,rA,rB	(OE = '1' Rc = '0')
addo.	rD,rA,rB	(OE = '1' Rc = '1')



$rD \leftarrow (rA) + (rB)$

The sum (rA) + (rB) is placed into rD.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

addc_x

Add Carrying (x'7C00 0014')

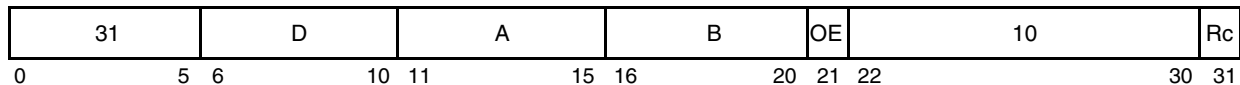
addc_x

addc **rD,rA,rB** (OE = '0' Rc = '0')

addc. **rD,rA,rB** (OE = '0' Rc = '1')

addco **rD,rA,rB** (OE = '1' Rc = '0')

addco. **rD,rA,rB** (OE = '1' Rc = '1')



$$rD \leftarrow (rA) + (rB)$$

The sum (rA) + (rB) is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

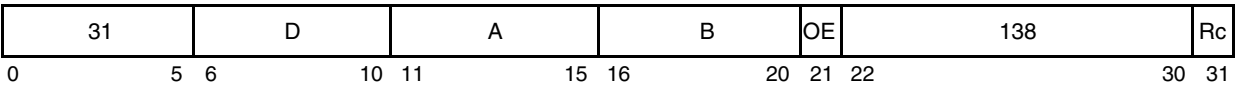
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				XO

addex

addex

Add Extended (x'7C00 0114')

adde	rD,rA,rB	(OE = '0' Rc = '0')
adde.	rD,rA,rB	(OE = '0' Rc = '1')
addeo	rD,rA,rB	(OE = '1' Rc = '0')
addeo.	rD,rA,rB	(OE = '1' Rc = '1')



$$rD \leftarrow (rA) + (rB) + XER[CA]$$

The sum (rA) + (rB) + XER[CA] is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

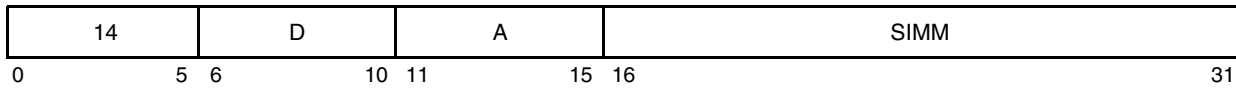
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

addi

Add Immediate (x'3800 0000')

addi

addi **rD,rA,SIMM**



```

if rA = 0 then
    rD ← EXTS (SIMM)
else
    rD ← (rA) + EXTS (SIMM)
    
```

The sum (rA10) + sign extended SIMM is placed into rD.

The **addi** instruction is preferred for addition because it sets few status bits.

Note: **addi** uses the value 0, not the contents of GPR0, if rA = '0'.

Other registers altered:

- None

Simplified mnemonics:

li	rD,value	equivalent to	addi rD,0,value
la	rD,disp(rA)	equivalent to	addi rD,rA,disp
subi	rD,rA,value	equivalent to	addi rD,rA,-value

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

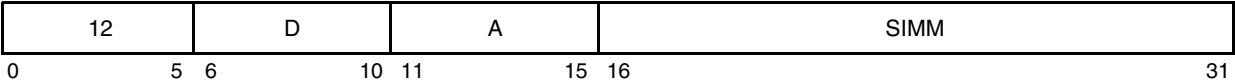


addic

addic

Add Immediate Carrying (x'3000 0000')

addic rD,rA,SIMM



$$rD \leftarrow (rA) + \text{EXTS}(SIMM)$$

The sum (rA) + sign extended SIMM is placed into rD.

Other registers altered:

- XER
Affected: CA

Note: For more information, see the XER Register section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

subic rD,rA,value equivalent to **addic** rD,rA,-value

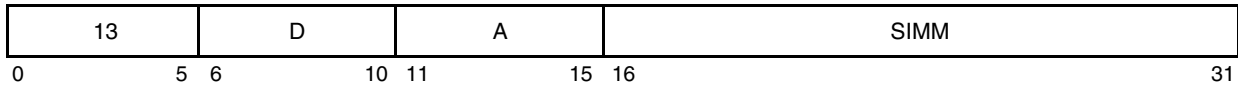
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

addic.

Add Immediate Carrying and Record (x'3400 0000')

addic.

addic. rD,rA,SIMM



$$rD \leftarrow (rA) + \text{EXTS}(SIMM)$$

The sum (rA) + the sign extended SIMM is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: CA

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

subic. rD,rA,value equivalent to **addic.** rD,rA,-value

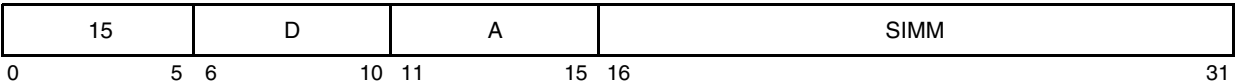
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

addis

addis

Add Immediate Shifted (x'3C00 0000')

addis rD,rA,SIMM



```
if rA = 0 then
    rD ← (SIMM || (16)0)
else
    rD ← (rA) + (SIMM || (16)0)
```

The sum (rA|0) + (SIMM || 0x0000) is placed into rD.

The **addis** instruction is preferred for addition because it sets few status bits.

Note: **addis** uses the value 0, not the contents of GPR0, if rA = '0'.

Other registers altered:

- None

Simplified mnemonics:

lis	rD,value	equivalent to	addis	rD,0,value
subis	rD,rA,value	equivalent to	addis	rD,rA,-value

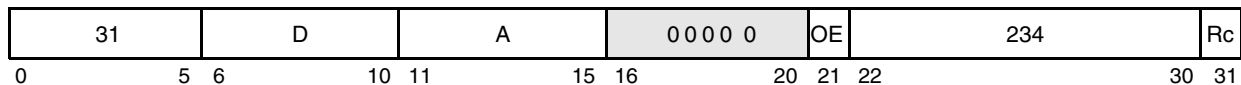
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

addmex

Add to Minus One Extended (x'7C00 01D4')

addmex

addme	rD,rA	(OE = '0' Rc = '0')
addme.	rD,rA	(OE = '0' Rc = '1')
addmeo	rD,rA	(OE = '1' Rc = '0')
addmeo.	rD,rA	(OE = '1' Rc = '1')

 Reserved


$$rD \leftarrow (rA) + XER[CA] - 1$$

The sum $(rA) + XER[CA] + 0xFFFF_FFFF$ is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER:
Affected: CA
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

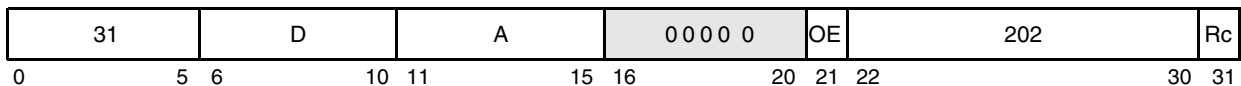
addze_x

Add to Zero Extended (x'7C00 0194')

addze_x

addze	rD,rA	(OE = '0' Rc = '0')
addze.	rD,rA	(OE = '0' Rc = '1')
addzeo	rD,rA	(OE = '1' Rc = '0')
addzeo.	rD,rA	(OE = '1' Rc = '1')

 Reserved



$$rD \leftarrow (rA) + XER[CA]$$

The sum (rA) + XER[CA] is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER:
Affected: CA
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

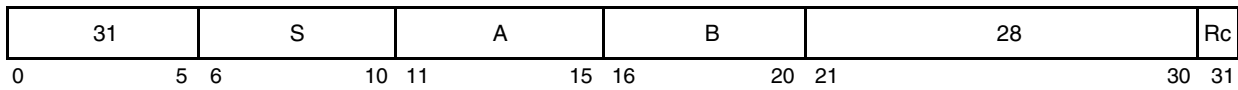
and_x

AND (x'7C00 0038')

and_x

and rA,rS,rB (Rc = '0')

and. rA,rS,rB (Rc = '1')



$$rA \leftarrow (rS) \& (rB)$$

The contents of rS are ANDed with the contents of rB, and the result is placed into rA.

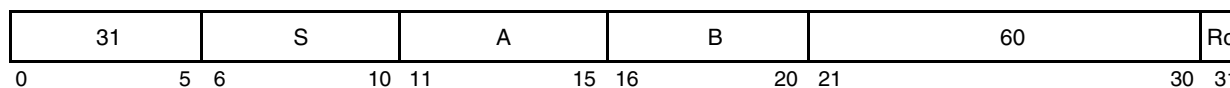
Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

andc_x

andc	rA,rS,rB	(Rc = '0')
andc.	rA,rS,rB	(Rc = '1')



$$\mathbf{rA} \leftarrow (\mathbf{rS}) \ \& \ \neg \ (\mathbf{rB})$$

Other registers altered:

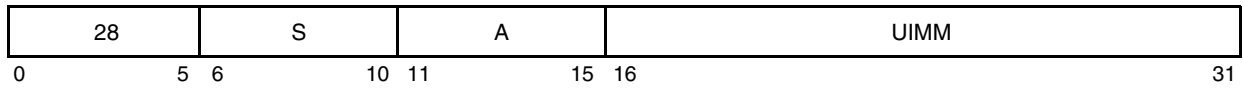
- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

andi.

AND Immediate (x'7000 0000')

andi. **rA,rS,UIMM**



```
rA ← (rS) & ((16)0 || UIMM)
```

The contents of **rS** are ANDed with 0x000 || UIMM, and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO

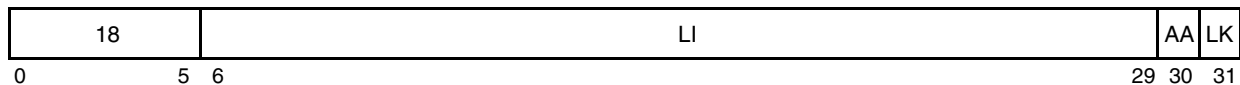
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

bx

bx

Branch (x'4800 0000')

b	target_addr	(AA = '0' LK = '0')
ba	target_addr	(AA = '1' LK = '0')
bl	target_addr	(AA = '0' LK = '1')
bla	target_addr	(AA = '1' LK = '1')



```

if AA = 1 then
    NIA ← iea EXTS(LI || 0b00)
else
    NIA ← iea CIA + EXTS(LI || 0b00)
if LK = 1 then
    LR ← iea CIA + 4

```

target_addr specifies the branch target address.

If AA = '1', the branch target address is the value LI || 0b00 sign-extended.

If AA = '0', the branch target address is the sum of LI || 0b00 sign-extended plus the address of this instruction.

If LK = '1', the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

- Link Register (LR) (if LK = '1')

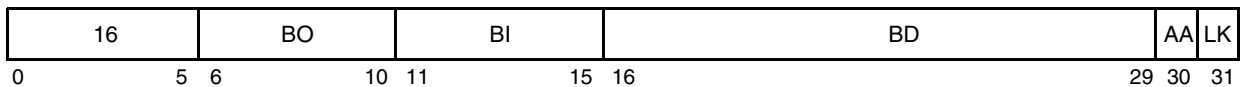
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				I

bc_x

Branch Conditional (x'4000 0000')

bc_x

bc	BO,BI,target_addr	(AA = '0' LK = '0')
bca	BO,BI,target_addr	(AA = '1' LK = '0')
bcl	BO,BI,target_addr	(AA = '0' LK = '1')
bcla	BO,BI,target_addr	(AA = '1' LK = '1')



```
if ¬ BO[2] then
    CTR ← CTR - 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
    if AA = 1 then
        NIA ←iea EXTS(BD || 0b00)
    else
        NIA ←iea CIA + EXTS(BD || 0b00)
if LK = 1 then
    LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register (CR) to be used as the condition of the branch. The BO field is encoded as described in *Table 10-6*. Additional information about BO field encoding is provided in the conditional branch control section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 10-6. BO Operand Encodings

BO	Description
0000y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR ≠ '0'.
1z01y	Decrement the CTR, then branch if the decremented CTR = '0'.
1z1zz	Branch always.

Note: In this table, z indicates a bit that is ignored. The z bits should be cleared, because they might be assigned a meaning in some future version of the PowerPC Architecture.
The y bit provides a hint about whether a conditional branch is likely to be taken, and might be used by some PowerPC implementations to improve performance.

**Advance****IBM Espresso RISC Microprocessor**

target_addr specifies the branch target address.

If AA = '0', the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA = '1', the branch target address is the value BD || 0b00 sign-extended.

If LK = '1', the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

- Count Register (CTR) (if BO[2] = '0')
- Link Register (LR) (if LK = '1')

Simplified mnemonics:

blt	target	equivalent to	bc 12,0,target
bne	cr2,target	equivalent to	bc 4,10,target
bdnz	target	equivalent to	bc 16,0,target

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

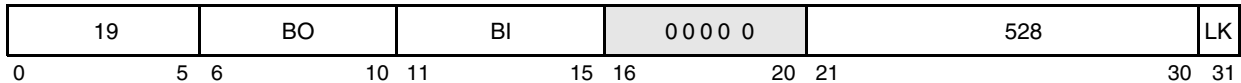
bcctr_x

bcctr_x

Branch Conditional to Count Register (x'4C00 0420')

bcctr BO,BI (LK = '0')
bcctrl BO,BI (LK = '1')

☐ Reserved



```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok
then
  NIA ←iea CTR || 0b00
  if LK then LR ←iea CIA + 4
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in *Table 10-7*. Additional information about BO field encoding is provided in the conditional branch control section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 10-7. BO Operand Encodings

BO	Description
0000y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR ≠ '0'.
1z01y	Decrement the CTR, then branch if the decremented CTR = '0'.
1z1zz	Branch always.

Note: In this table, z indicates a bit that is ignored. The z bits should be cleared, because they might be assigned a meaning in some future version of the PowerPC Architecture.

The y bit provides a hint about whether a conditional branch is likely to be taken, and might be used by some PowerPC implementations to improve performance.

The branch target address is CTR[0:29] || '00'.
 If LK = '1', the effective address of the instruction following the branch instruction is placed into the link register.
 If the “decrement and test CTR” option is specified (BO[2] = '0'), the instruction form is invalid.

**Advance****IBM Espresso RISC Microprocessor**

Other registers altered:

- Link Register (LR) (if LK = '1')

Simplified mnemonics:

bltctr equivalent to **bcctr** 12,0
bnectr cr2 equivalent to **bcctr** 4,10

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				XL

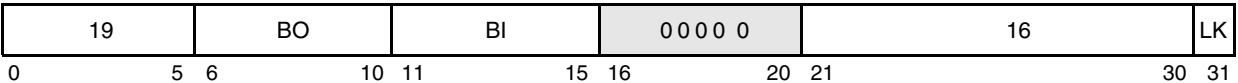
bclr_x

bclr_x

Branch Conditional to Link Register (x'4C00 0020')

bclr BO,BI (LK = '0')
bclrl BO,BI (LK = '1')

Reserved



```

if ¬ BO[2]
    then CTR ← CTR - 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok
    then NIA ←iea LR[0-29] || 0b00
if LK
    then LR ←iea CIA + 4
    
```

The BI field specifies the bit in the condition register to be used as the condition of the branch. The BO field is encoded as described in *Table 10-8*. Additional information about BO field encoding is provided in the conditional branch control section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Table 10-8. BO Operand Encodings

BO	Description
0000y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is FALSE.
0001y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is FALSE.
001zy	Branch if the condition is FALSE.
0100y	Decrement the CTR, then branch if the decremented CTR ≠ '0' and the condition is TRUE.
0101y	Decrement the CTR, then branch if the decremented CTR = '0' and the condition is TRUE.
011zy	Branch if the condition is TRUE.
1z00y	Decrement the CTR, then branch if the decremented CTR ≠ '0'.
1z01y	Decrement the CTR, then branch if the decremented CTR = '0'.
1z1zz	Branch always.

Notes:
 If the BO field specifies that the CTR is to be decremented, the entire 32-bit CTR is decremented.
 In this table, z indicates a bit that is ignored. The z bits should be cleared, as they might be assigned a meaning in some future version of the PowerPC Architecture.
 The y bit provides a hint about whether a conditional branch is likely to be taken, and might be used by some PowerPC implementations to improve performance.

The branch target address is LR[0:29] || 0b00.

If LK = '1', the effective address of the instruction following the branch instruction is placed into the Link Register.

**Advance****IBM Espresso RISC Microprocessor**

Other registers altered:

- Count Register (CTR) (if BO[2] = '0')
- Link Register (LR) (if LK = '1')

Simplified mnemonics:

bltlr	equivalent to	bclr 12,0
bnelr cr2	equivalent to	bclr 4,10
bdnzlr	equivalent to	bclr 16,0

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				XL



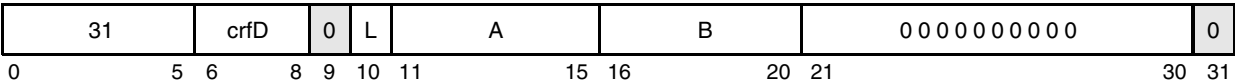
cmp

Compare (x'7C00 0000')

cmp crfD,L,rA,rB

cmp

 Reserved



```
a ← (rA)
b ← (rB)
if a < b then
  c ← 0b100
else if a > b then
  c ← 0b010
else
  c ← 0b001
CR[ (4 × crfD) - (4 × crfD + 3) ] ← c || XER[SO]
```

The contents of rA are compared with the contents of rB, treating the operands as signed integers. The result of the comparison is placed into CR field crfD.

If L = '1', the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand crfD)
Affected: LT, GT, EQ, SO

Simplified mnemonics:

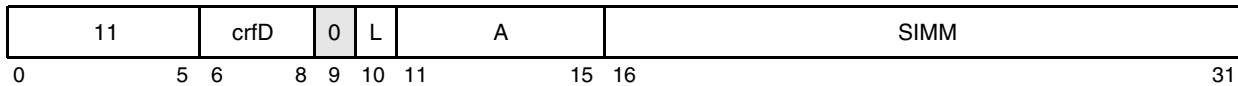
cmpd rA,rB equivalent to cmp 0,1,rA,rB
cmpw cr3,rA,rB equivalent to cmp 3,0,rA,rB

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

cmpi

Compare Immediate (x'2C00 0000')

cmpi

cmpi
crfD,L,rA,SIMM
☐ Reserved


```

a ← (rA)
if a < EXTS(SIMM) then
    c ← 0b100
else if a > EXTS(SIMM) then
    c ← 0b010
else
    c ← 0b001
CR[(4 × crfD) - (4 × crfD + 3)] ← c || XER[SO]
    
```

The contents of **rA** are compared with the sign-extended value of the SIMM field, treating the operands as signed integers. The result of the comparison is placed into CR field **crfD**.

If L = '1', the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpdi	rA,value	equivalent to	cmpi 0,1,rA,value
cmpwi	cr3,rA,value	equivalent to	cmpi 3,0,rA,value

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

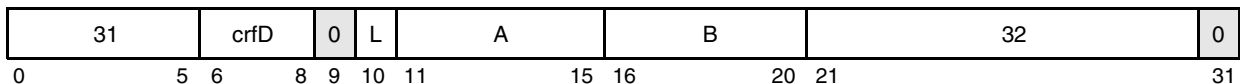
cmpl

Compare Logical (x'7C00 0040')

cmpl

cmpl **crfD,L,rA,rB**

 Reserved



```
a ← (rA)
b ← (rB)
if a <U b then
    c ← 0b100
else if a >U b then
    c ← 0b010
else
    c ← 0b001
CR[ (4 × crfD) - (4 × crfD + 3) ] ← c || XER[SO]
```

The contents of **rA** are compared with the contents of **rB**, treating the operands as unsigned integers. The result of the comparison is placed into the CR field **crfD**.

If L = '1', the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpld	rA,rB	equivalent to	cmpl	0,1,rA,rB
cmplw	cr3,rA,rB	equivalent to	cmpl	3,0,rA,rB

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

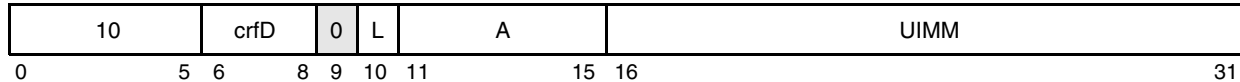
cmpli

Compare Logical Immediate (x'2800 0000')

cmpli

cmpli **crfD,L,rA,UIMM**

 Reserved



```

a ← (rA)
if a <U ((16)0 || UIMM) then
    c ← 0b100
else if a >U ((16)0 || UIMM) then
    c ← 0b010
else
    c ← 0b001
CR[(4 × crfD) - (4 × crfD + 3)] ← c || XER[SO]

```

The contents of **rA** are compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crfD**.

If L = '1', the instruction form is invalid.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, SO

Simplified mnemonics:

cmpldi	r A,value	equivalent to	cmpli	0,1,rA,value
cmplwi	cr3,rA,value	equivalent to	cmpli	3,0,rA,value

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

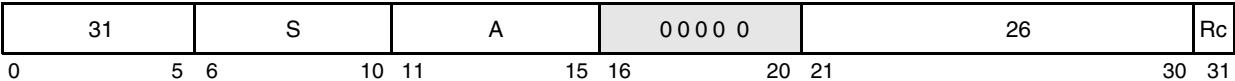
cntlzw_x

Count Leading Zeros Word (x'7C00 0034')

cntlzw_x

cntlzw rA,rS (Rc = '0')
cntlzw. rA,rS (Rc = '1')

☐ Reserved



```
n ← 0
do while n < 32
    if rS[n] = 1 then leave
    n ← n + 1
rA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of **rS** is placed into **rA**. This number ranges from 0 to 32, inclusive.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: If Rc = '1', LT is cleared in the CR0 field.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

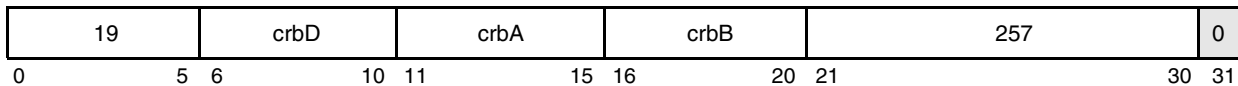
crand

Condition Register AND (x'4C00 0202')

crand

crand **crbD,crbA,crbB**

 Reserved



$CR[crbD] \leftarrow CR[crbA] \& CR[crbB]$

The bit in the condition register specified by **crbA** is ANDed with the bit in the condition register specified by **crbB**, and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

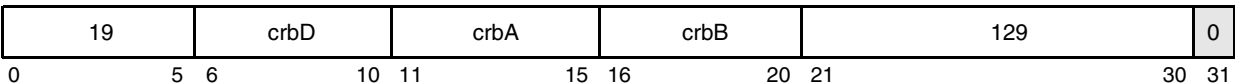
crandc

crandc

Condition Register AND with Complement (x'4C00 0102')

crandc **crbD,crbA,crbB**

 Reserved



$CR[\text{crbD}] \leftarrow CR[\text{crbA}] \ \& \ \neg CR[\text{crbB}]$

The bit in the condition register specified by **crbA** is ANDed with the complement of the bit in the condition register specified by **crbB**, and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

creqv

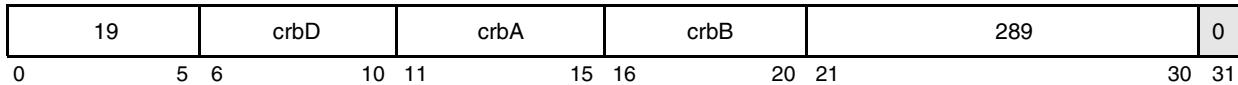
Condition Register Equivalent (x'4C00 0242')

creqv

creqv

crbD,crbA,crbB

☐ Reserved



$$CR[\text{crbD}] \leftarrow CR[\text{crbA}] \oplus CR[\text{crbB}]$$

The bit in the condition register specified by **crbA** is XORed with the bit in the condition register specified by **crbB**. The complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

Simplified mnemonics:

crse **crbD** equivalent to **creqv** **crbD,crbD,crbD**

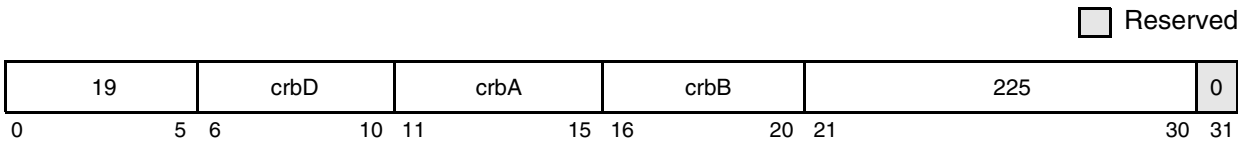
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

crnand

Condition Register NAND (x'4C00 01C2')

crnand

crbD,crbA,crbB



$$CR[crbD] \leftarrow \neg (CR[crbA] \& CR[crbB])$$

The bit in the condition register specified by **crbA** is ANDed with the bit in the condition register specified by **crbB**. The complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

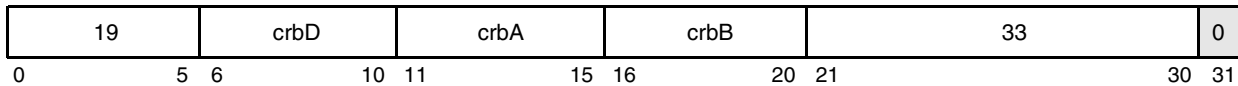
crnor

Condition Register NOR (x'4C00 0042')

crnor

crnor **crbD,crbA,crbB**

 Reserved



$CR[crbD] \leftarrow \neg (CR[crbA] \mid CR[crbB])$

The bit in the condition register specified by **crbA** is ORed with the bit in the condition register specified by **crbB**, and the complemented result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

Simplified mnemonics:

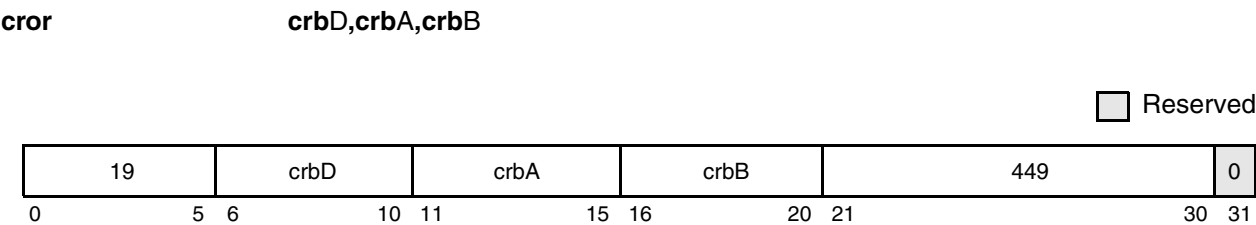
crnot **crbD,crbA** equivalent to **crnor** **crbD,crbA,crbA**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

cror

Condition Register OR (x'4C00 0382')

cror



CR[crbD] ← CR[crbA] | CR[crbB]

The bit in the condition register specified by **crbA** is ORed with the bit in the condition register specified by **crbB**, and the result is placed into the condition register bit specified by **crbD**.

- Other registers altered:
- Condition Register
Affected: Bit specified by operand **crbD**

Simplified mnemonics:

crmove **crbD,crbA** equivalent to **cror** **crbD,crbA,crbA**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

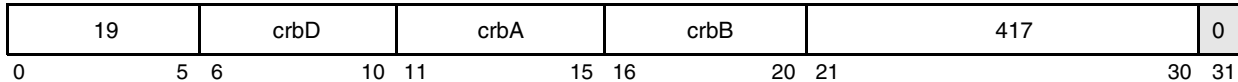
crorc

crorc

Condition Register OR with Complement (x'4C00 0342')

crorc **crbD,crbA,crbB**

 Reserved



$CR[crbD] \leftarrow CR[crbA] \mid \neg CR[crbB]$

The bit in the condition register specified by **crbA** is ORed with the complement of the condition register bit specified by **crbB**, and the result is placed into the condition register bit specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by operand **crbD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL



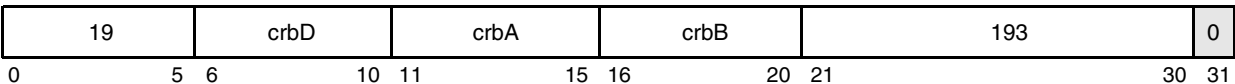
crxor

Condition Register XOR (x'4C00 0182')

crxor

crxor **crbD,crbA,crbB**

 Reserved



$$CR[\text{crbD}] \leftarrow CR[\text{crbA}] \oplus CR[\text{crbB}]$$

The bit in the condition register specified by **crbA** is XORed with the bit in the condition register specified by **crbB**, and the result is placed into the condition register specified by **crbD**.

Other registers altered:

- Condition Register
Affected: Bit specified by **crbD**

Simplified mnemonics:

crclr **crbD** equivalent to **crxor** **crbD,crbD,crbD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XL

dcbf

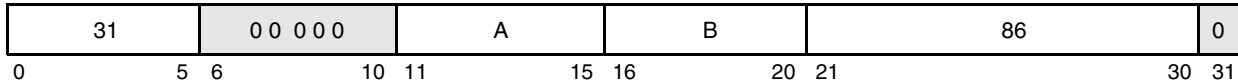
Data Cache Block Flush (x'7C00 00AC')

dcbf

dcbf

rA,rB

☐ Reserved



EA is the sum (rA10) + (rB).

The action taken depends on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. If the block is marked coherency-required, the processor will, if necessary, send an address-only broadcast to other processors. The broadcast of the **dcbf** instruction causes another processor to copy the block to memory if it has dirty data, and then invalidate the block from the cache. The following list describes the action taken for the two states of the memory coherency attribute (M bit).

- Coherency required (requires the use of address broadcast)
 - Unmodified block. Invalidates copies of the block in the data caches of all processor.
 - Modified block. Copies the block to memory and invalidates it. (In what ever processor it resides, there should be only one modified block)
 - Absent block. A modified copy of the block is in the data cache of another processor, causes that processor to copy the block to memory and invalidates it in its data cache. If unmodified copies are in the data caches of other processors, those copies are invalidated in those data caches.
- Coherency not required (no address broadcast required)
 - Unmodified block. Invalidates the block in the processor data cache.
 - Modified block. Copies the block to memory. Invalidates the block in the processor data cache.
 - Absent block. No action is taken.

The function of this instruction is independent of the write-through, write-back, and caching-inhibited/allowed modes of the block containing the byte addressed by the EA. This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording, except that referenced and changed bit recording might not occur.

When HID2[LCE] = '1' and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation/push, or forwarded to the 60x bus for broadcast. Otherwise, the instruction is forwarded to the L2 cache and to the 60x bus as described in the the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

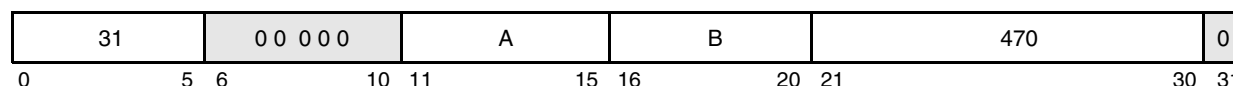
dcbi

Data Cache Block Invalidate (x'7C00 03AC')

dcbi

dcbi

rA,rB

 Reserved

EA is the sum (rA10) + (rB).

The action taken is dependent on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The following list describes the action taken if the block containing the byte addressed by EA is or is not in the cache.

- Coherency required (requires the use of address broadcast)
 - Unmodified block. Invalidates copies of the block in the data caches of all processor.
 - Modified block. Invalidates the copy of the block in the data cache in the processors where it is found. (Discards the modified contents).
 - Absent block. A modified copy of the block in the data cache of another processor, causes that processor to invalidate it in its data cache. If unmodified copies are in the data caches of other processors, those copies are invalidated in those data caches.
- Coherency not required (no address broadcast required)
 - Unmodified block. Invalidates the block in the processor data cache.
 - Modified block. Invalidates the block in the processor data cache. (Discards the modified contents.)
 - Absent block (target block not in cache). No action is taken.

When data address translation is enabled, MSR[DR] = '1', and the virtual address has no translation, a DSI exception occurs.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA. This instruction operates as a store to the addressed byte with respect to address translation and protection. The referenced and changed bits are modified appropriately.

When HID2[LCE] = '1' and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation, or forwarded to the 60x bus for broadcast. Otherwise, the instruction is forwarded to the L2 cache and to the 60x bus as described in the the *PowerPC Microprocessor Family: The Programming Environments* manual. This is a supervisor-level instruction by default. However, this can be made a user-level instruction.



Other registers altered:

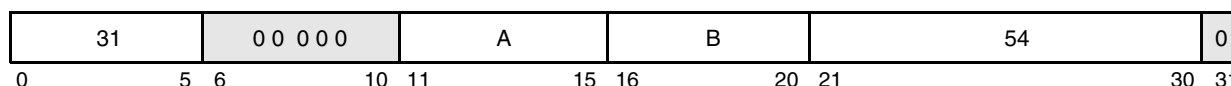
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA	Yes ¹			X
1. When instruction is a supervisor-level instruction, value is Yes.				

dcbst

Data Cache Block Store (x'7C00 006C')

dcbst

dcbst**rA,rB** Reserved

EA is the sum (rA10) + (rB).

The **dcbst** instruction executes as follows:

- Coherency required (requires the use of address broadcast).
 - Unmodified block. No action in this processor. Signals other processors to copy to memory any modified cache block.
 - Modified block. The cache block is written to memory. (Only one processor can have a copy of a modified block.)
 - Absent block. No action in this processor. If a modified copy of the block is in the data cache of another processor, the cache line is written to memory.
- Coherency not required (no address broadcast required).
 - Unmodified block. No action is taken.
 - Modified block. The cache block is written to memory.
 - Absent block. No action is taken.

The function of this instruction is independent of the write-through and caching inhibited/allowed modes of the block containing the byte addressed by EA.

The processor treats this instruction as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording except that referenced and changed bit recording might not occur.

When HID2[LCE] = '1' and the byte addressed by EA is in the locked cache, the instruction is not forwarded to the L2 cache for sector invalidation/push or forwarded to the 60X bus for broadcast. Otherwise, the instruction is forwarded to the L2 cache and to the 60x bus.

Other registers altered:

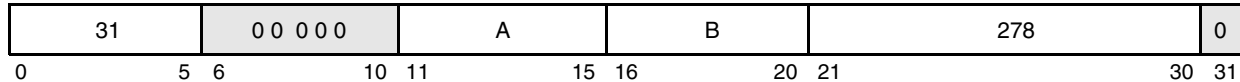
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

dcbt

Data Cache Block Touch (x'7C00 022C')

dcbt

dcbt**rA,rB**☐ Reserved

EA is the sum (rA10) + (rB).

This instruction is a hint that performance might be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program might load soon from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbt** does not cause the system alignment error handler to be invoked.

If HID2[LCE] = '1' and the byte addressed by EA is in neither the locked nor the normal cache, this instruction loads the cache line into the "normal" cache.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that referenced and changed bit recording might not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses the **dcbt** instruction to request a cache block fetch before it is actually needed by the program. The program can later execute load instructions to put data into registers. However, the processor is not required to load the addressed block into the data cache.

Note: This instruction is defined architecturally to perform the same functions as the **dcbtst** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

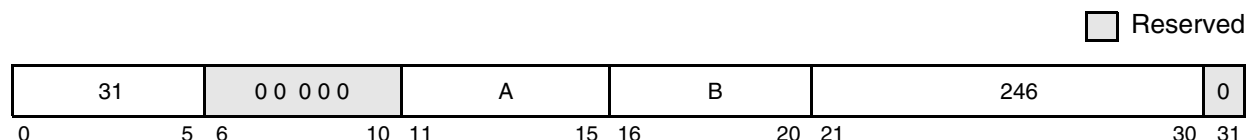
dcbtst

Data Cache Block Touch for Store (x'7C00 01EC')

dcbtst

 r_A, r_B

dcbtst



EA is the sum $(\mathbf{r}_A|0) + (\mathbf{r}_B)$.

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbtst** does not cause the system alignment error handler to be invoked.

If **HID2[LCE]** = '1' and the byte addressed by **EA** is in neither the locked nor the normal cache, this instruction loads the cache line into the normal cache.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording, except that referenced and changed bit recording might not occur. Additionally, no exception occurs in the case of a translation fault or protection violation.

The program uses **dcbtst** to request a cache block fetch to potentially improve performance for a subsequent store to that EA, because store would then be to a cached location. However, the processor is not obliged to load the addressed block into the data cache.

Note: This instruction is defined architecturally to perform the same functions as the **dcbt** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

dcbz

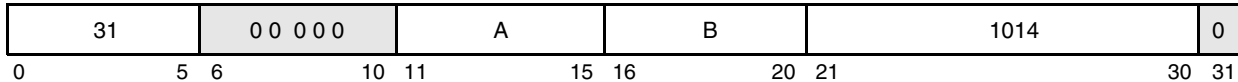
Data Cache Block Clear to Zero (x'7C00 07EC')

dcbz

dcbz

rA,rB

☐ Reserved



EA is the sum (rA10) + (rB).

This instruction is treated as a store to the addressed byte with respect to address translation, memory protection, referenced and changed recording. It is also treated as a store with respect to the ordering enforced by **ei** and the ordering enforced by the combination of caching-inhibited and guarded attributes for a page (or block).

The **dcbz** instruction executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, all bytes are cleared and the cache line is marked "M".
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding memory page or block is caching-allowed, the cache block is allocated (and made valid) in the data cache (or in the normal cache if HID2[LCE] = '1') without fetching the block from main memory, and all bytes are cleared.
- If the page containing the byte addressed by EA is in caching-inhibited or write-through mode, either all bytes of main memory that correspond to the addressed cache block are cleared or the alignment exception handler is invoked. The exception handler can then clear all bytes in main memory that correspond to the addressed cache block.
- If the cache block containing the byte addressed by EA is in coherency-required mode, and the cache block exists in the data cache of any other processor, it is kept coherent in that cache (that is, the processor performs the appropriate bus transactions to enforce this).

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

dcbz |

dcbz I **rA,rB**



When `HID2[LCE] = '1'`, the **dcbz** I instruction executes as follows:

- If the cache block containing the byte addressed by EA is neither in the locked nor in the normal data cache, the block is allocated in the locked data cache without fetching the block from main memory. All bytes are cleared and the block is marked as modified (M). Cache block allocation is done using the pseudo-LRU used rule among the four ways in the locked cache.
- If the cache block containing the byte addressed by EA is already either in the locked or in the normal data cache, all bytes are cleared and the block is marked modified (M). The hardware indicates this situation by setting `HID2[DCHERR] = '1'` and raising a machine check condition.
- The **dcbz** instruction is not forwarded to the L2 cache nor the 60Xe bus for broadcast.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA		Yes		X

divw_x

Divide Word (x'7C00 03D6')

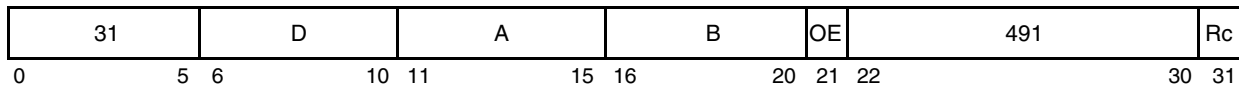
divw_x

divw **rD,rA,rB** (OE = '0' Rc = '0')

divw. **rD,rA,rB** (OE = '0' Rc = '1')

divwo **rD,rA,rB** (OE = '1' Rc = '0')

divwo. **rD,rA,rB** (OE = '1' Rc = '1')



```
dividend ← (rA)
divisor ← rB
rD ← dividend / divisor
```

The dividend is the contents of **rA**. The divisor is the contents of **rB**. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation:

dividend = (quotient × divisor) + r where $0 \leq r < |\text{divisor}|$ (if the dividend is nonnegative), and $-|\text{divisor}| < r \leq 0$ (if the dividend is negative).

If an attempt is made to perform either of the divisions:

0x8000_0000 ÷ -1 or
<anything> ÷ 0

then the contents of **rD** are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = '1'). In this case, if OE = '1', OV is set.

The 32-bit signed remainder of dividing the contents of **rA** by the contents of **rB** can be computed as follows, except in the case that the contents of **rA** = -2³¹ and the contents of **rB** = -1.

divw **rD,rA,rB** # rD = quotient

mullw **rD,rD,rB** # rD = quotient × divisor

subf **rD,rD,rA** # rD = remainder

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')
- XER
Affected: SO, OV (if OE = '1')

Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

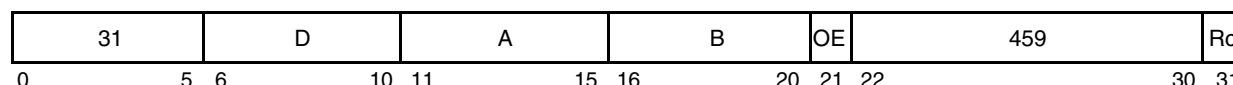
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

divwux

Divide Word Unsigned (x'7C00 0396')

divwux

divwu	rD,rA,rB	(OE = '0' Rc = '0')
divwu.	rD,rA,rB	(OE = '0' Rc = '1')
divwuo	rD,rA,rB	(OE = '1' Rc = '0')
divwuo.	rD,rA,rB	(OE = '1' Rc = '1')



```

dividend ← (rA)
divisor ← (rB)
rD ← dividend ÷ divisor

```

The dividend is the contents of **rA**. The divisor is the contents of **rB**. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if **Rc** = '1' the first three bits of **CR0** field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation:

$$\text{dividend} = (\text{quotient} \times \text{divisor}) + r \text{ (where } 0 \leq r < \text{divisor}).$$

If an attempt is made to perform the division: $\text{<anything>} \div 0$, then the contents of **rD** are undefined as are the contents of the **LT**, **GT**, and **EQ** bits of the **CR0** field (if **Rc** = '1'). In this case, if **OE** = '1', **OV** is set.

The 32-bit unsigned remainder of dividing the contents of **rA** by the contents of **rB** can be computed as follows:

divwu	rD,rA,rB	# rD = quotient
mullw	rD,rD,rB	# rD = quotient × divisor
subf	rD,rD,rA	# rD = remainder

Other registers altered:

- Condition Register (CR0 field)
Affected: **LT**, **GT**, **EQ**, **SO** (if **Rc** = '1')
- XER**
Affected: **SO**, **OV** (if **OE** = '1')

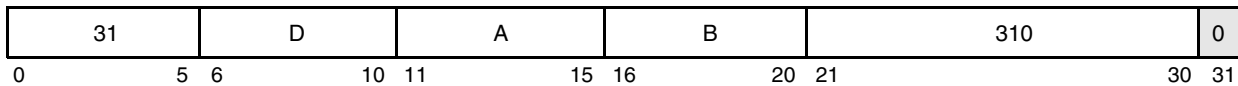
Note: For more information about condition codes, see the Condition Register and XER Register sections in the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

eciwx

External Control In Word Indexed (x'7C00 026C')

eciwx

eciwx **rD,rA,rB** Reserved

The **eciwx** instruction and the External Access Register (EAR) can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send load word request for paddr to device identified by EAR[RID]
rD ← word from device

```

EA is the sum (rA|0) + (rB).

A load word request for the physical address (referred to as real address in the architecture specification) corresponding to EA is sent to the device identified by EAR[RID], bypassing the cache. The word returned by the device is placed in rD.

EAR[E] must be '1'. If it is not, a DSI exception is generated. EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = '0').
- The results are boundedly undefined.

The **eciwx** instruction is supported for EAs that reference memory segments in which SR[T] = '1' (or STE[T] = '1') and for EAs mapped by the DBAT registers. If the EA references a direct-store segment (SR[T] = '1' or STE[T] = '1'), either a DSI exception occurs or the results are boundedly undefined.

Note: The direct-store facility is being phased out of the architecture and might not be supported in future devices. Thus, software should not depend on its effects.

If this instruction is executed when MSR[DR] = '0' (real addressing mode), the results are boundedly undefined. This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, referenced and changed bit recording, and the ordering performed by **eieio**.

This instruction is optional in the PowerPC Architecture.

Other registers altered:

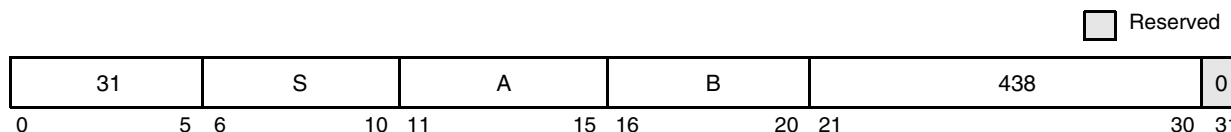
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA			YES	X

ecowx

ecowx

External Control Out Word Indexed (x'7C00 036C')

ecowx**rS,rA,rB**

The **ecowx** instruction and the EAR Register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send store word request for paddr to device identified by EAR[RID]
send rS to device

```

EA is the sum (rA10) + (rB).

A store word request for the physical address corresponding to EA and the contents of rS are sent to the device identified by EAR[RID], bypassing the cache. EAR[E] must be '1'; if it is not, a DSI exception is generated. EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment exception is generated.
- A DSI exception is generated (possible only if EAR[E] = '0').
- The results are boundedly undefined.

The **ecowx** instruction is supported for effective addresses that reference memory segments in which SR[T] = '0' or STE[T] = '0', and for EAs mapped by the DBAT registers. If the EA references a direct-store segment (SR[T] = '1' or STE[T] = '1'), either a DSI exception occurs or the results are boundedly undefined.

Note: The direct-store facility is being phased out of the architecture and might not be supported in future devices. Thus, software should not depend on its effects.

If this instruction is executed when MSR[DR] = '0' (real addressing mode), the results are boundedly undefined. This instruction is treated as a store from the addressed byte with respect to address translation, memory protection, referenced and changed bit recording, and the ordering performed by **eieio**.

Note: Software synchronization is required to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded.

This instruction is optional in the PowerPC Architecture

Other registers altered:

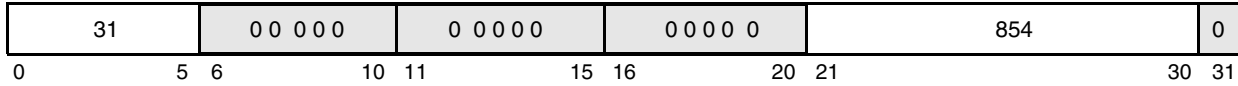
- None

eieio

Enforce In-Order Execution of I/O (x'7C00 06AC')

eieio

☐ Reserved



The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The memory accesses caused by a **dcbz** or a **dcba** instruction are ordered like a store. The two sets follow:

1. Loads and stores to memory that is both caching-inhibited and guarded, and stores to memory that is write-through required.

The **eieio** instruction controls the order in which the accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **eieio** instruction have completed with respect to main memory before any applicable memory accesses caused by instructions following the **eieio** instruction access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.

All accesses in this set are ordered as a single set; that is, there is not one order for loads and stores to caching-inhibited and guarded memory and another order for stores to write-through required memory.

2. Stores to memory that have all of the following attributes: caching-allowed, write-through not required, and memory-coherency required.

The **eieio** instruction controls the order in which the accesses are performed with respect to coherent memory. It ensures that all applicable stores caused by instructions preceding the **eieio** instruction have completed with respect to coherent memory before any applicable stores caused by instructions following the **eieio** instruction complete with respect to coherent memory.

With the exception of **dcbz** and **dcba**, **eieio** does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction, or implicitly by the cache coherency mechanism). For more information, see the cache model and memory coherency section of the *PowerPC Microprocessor Family: The Programming Environments* manual. The **eieio** instruction does not affect the order of accesses in one set with respect to accesses in the other set.

The **eieio** instruction can complete before memory accesses caused by instructions preceding the **eieio** instruction have been performed with respect to main memory or coherent memory as appropriate.

The **eieio** instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (set 1 or set 2).

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA			YES	X

For the second use, **eieio** can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

Because the processor performs store operations to memory that is designated as both caching-inhibited and guarded (see the memory access ordering section in the *PowerPC Microprocessor Family: The Programming Environments* manual), the **eieio** instruction is required for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note: The **eieio** instruction does not connect hardware considerations to it, such as multiprocessor implementations, that send an **eieio** address-only broadcast (useful in some designs). For example, if a design has an external buffer that reorders loads and stores for better bus efficiency, the **eieio** broadcast signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

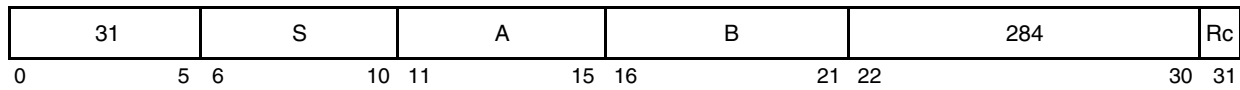
eqv_x

Equivalent (x'7C00 0238')

eqv_x

eqv rA,rS,rB (Rc = '0')

eqv. rA,rS,rB (Rc = '1')



$$\mathbf{rA} \leftarrow (\mathbf{rS}) \equiv (\mathbf{rB})$$

The contents of **rS** are XORed with the contents of **rB**, and the complemented result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

extsb_x

Extend Sign Byte (x'7C00 0774')

extsb_x

extsb

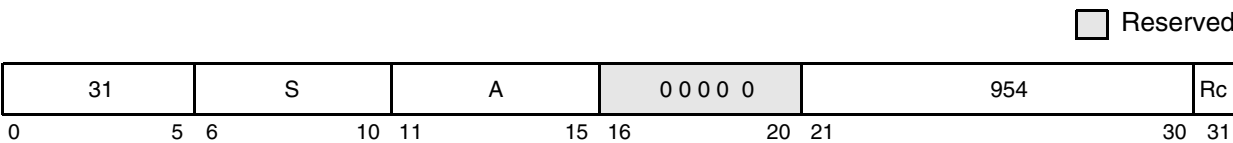
rA,rS

(Rc = '0')

extsb.

rA,rS

(Rc = '1')



$S \leftarrow rS[24]$

$rA[24-31] \leftarrow rS[24-31]$

$rA[0-23] \leftarrow (24)S$

The contents of the low-order 8 bits of **rS** are placed into the low-order 8 bits of **rA**.

Bit 24 of **rS** is placed into the remaining bits of **rA**.

- Other registers altered:
- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

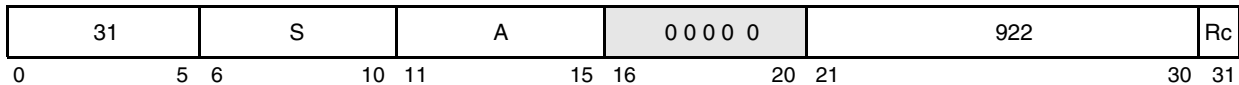
extsh_x

Extend Sign Halfword (x'7C00 0734')

extsh_x

extsh **rA,rS** (**Rc = '0'**)
extsh. **rA,rS** (**Rc = '1'**)

 Reserved



$S \leftarrow rS[16]$
 $rA[16-31] \leftarrow rS[16-31]$
 $rA[0-15] \leftarrow (16)S$

The contents of the low-order 16 bits of **rS** are placed into the low-order 16 bits of **rA**. Bit 16 of **rS** is placed into the remaining bits of **rA**.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if **Rc = '1'**)

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

fabs_x

Floating Absolute Value (x'FC00 0210')

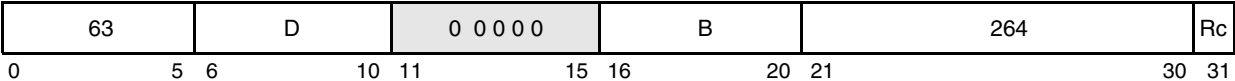
fabs_x

fabs
fabs.

frD,frB
frD,frB

(Rc = '0')
(Rc = '1')

Reserved



The contents of **frB** with bit 0 cleared are placed into **frD**.

Note: The **fabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN can be altered by **fabs**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

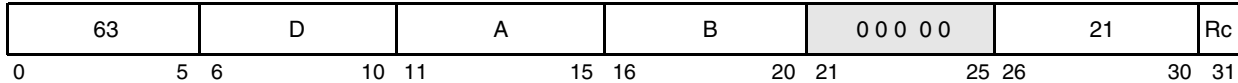
fadd_x

Floating Add (Double-Precision) (x'FC00 002A')

fadd_x

fadd **frD,frA,frB** (Rc = '0')

fadd. **frD,frA,frB** (Rc = '1')

☐ Reserved


The following operation is performed:

$$\mathbf{frD} \leftarrow (\mathbf{frA}) + (\mathbf{frB})$$

The floating-point operand in **frA** is added to the floating-point operand in **frB**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by 1 for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand, as well as all 3 guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right 1-bit position and the exponent is increased by 1. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

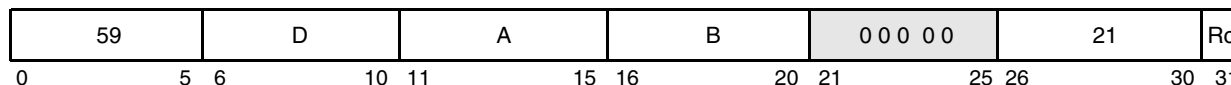
fadds_x

Floating Add Single (x'EC00 002A')

fadds_x

fadds **frD,frA,frB** (Rc = '0')

fadds. **frD,frA,frB** (Rc = '1')

☐ Reserved


The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← (frA) + (frB)
else
    frD_ps0 ← (frA_ps0) + (frB_ps0)
    frD_ps1 ← (frA_ps0) + (frB_ps0)

```

The floating-point operand in **frA** is added to the floating-point operand in **frB**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to the single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by 1 for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand, as well as all 3 guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right 1 bit position and the exponent is increased by 1. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If the HID2[PSE] = '1', the sum is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXIS

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fcmpo

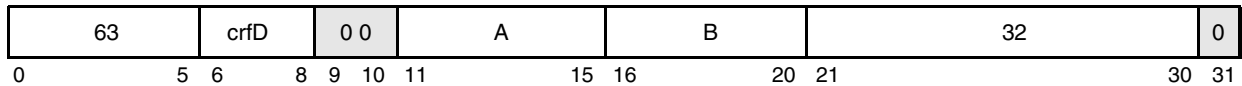
Floating Compare Ordered (x'FC00 0040')

fcmpo

fcmpo

crfD,frA,frB

Reserved



```

if ((frA) is a NaN or (frB) is a NaN) then
    c ← 0b0001
else if (frA) < (frB) then
    c ← 0b1000
else if (frA) > (frB) then
    c ← 0b0100
else
    c ← 0b0010
FPCC ← c
CR[(4 × crfD) - (4 × crfD + 3)] ← c
if ((frA) is an SNaN or (frB) is an SNaN ) then
    VXSNaN ← 1
if VE = 0 then
    VXVC ← 1
else if ((frA) is a QNaN or (frB) is a QNaN ) then
    VXVC ← 1
    
```

The floating-point operand in **frA** is compared to the floating-point operand in **frB**. The result of the compare is placed into CR field **crfD** and the floating-point condition code (FPCC).

If one of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, the VXSNaN is set. If invalid operation is disabled (VE = '0'), the VXVC is set. Otherwise, if one of the operands is a QNaN, the VXVC is set.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC, FX, VXSNaN, VXVC

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

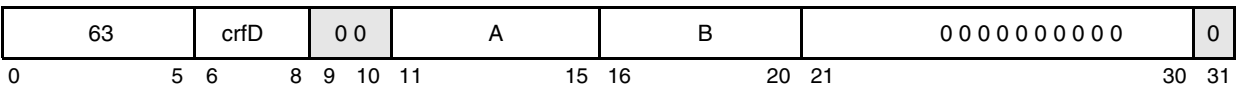
fcmphu

Floating Compare Unordered (x'FC00 0000')

fcmphu

fcmphu **crfD,frA,frB**

 Reserved



```
if ((frA) is a NaN or (frB) is a NaN) then
    c ← 0b0001
else if (frA) < (frB) then
    c ← 0b1000
else if (frA) > (frB) then
    c ← 0b0100
else
    c ← 0b0010
FPCC ← c
CR[(4 × crfD) - (4 × crfD + 3)] ← c

if ((frA) is an SNaN or (frB) is an SNaN) then
    VXSNaN ← 1
```

The floating-point operand in register **frA** is compared to the floating-point operand in register **frB**. The result of the compare is placed into CR field **crfD** and the FPCC.

If one of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, the VXSNaN is set.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC, FX, VXSNaN

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

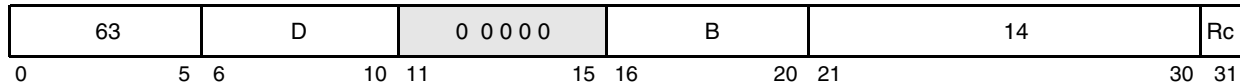
fctiw_x

Floating Convert to Integer Word (x'FC00 001C')

fctiw_x

fctiw **frD,frB** (Rc = '0')
fctiw. **frD,frB** (Rc = '1')

☐ Reserved



The floating-point operand in register **frB** is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in bits 32:63 of **frD**. Bits 0:31 of **frD** are undefined.

If the operand in **frB** are greater than $2^{31} - 1$, bits 32:63 of **frD** are set to x'7FFF_FFFF'.

If the operand in **frB** are less than -2^{31} , bits 32:63 of **frD** are set to x'8000_0000'.

The conversion is described in the floating-point convert to integer model section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Do not use this instruction if the floating-point register contains paired-single formatted data.

Programmers note: A **stfiwx** instruction should be used to store the 32-bit resultant integer.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (undefined), FR, FI, FX, XX, VXSNaN, VXCvI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				X

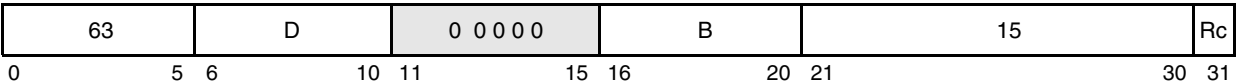
fctiwzx

fctiwzx

Floating Convert to Integer Word with Round toward Zero (x'FC00 001E')

fctiwz **frD,frB** (Rc = '0')
fctiwz. **frD,frB** (Rc = '1')

Reserved



The floating-point operand in register **frB** is converted to a 32-bit signed integer, using the rounding mode round toward zero, and placed in bits 32:63 of **frD**. Bits 0:31 of **frD** are undefined.

If the operand in **frB** is greater than $2^{31} - 1$, bits 32:63 of **frD** are set to x'7FFF_FFFF'.

If the operand in **frB** is less than -2^{31} , bits 32:63 of **frD** are set to x'8000_0000'.

The conversion is described in the floating-point convert to integer model section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Except for trap-enabled invalid operation exceptions, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Do not use this instruction if the floating-point register contains paired-single formatted data.

Programmers Note: A **stfiwx** instruction should be used to store the 32-bit resultant integer.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

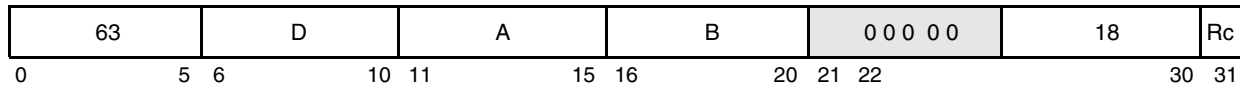
fdiv_x

fdiv_x

Floating Divide (Double-Precision), (x'FC00 0024')

fdiv **frD,frA,frB** (Rc = '0')
fdiv. **frD,frA,frB** (Rc = '1')

 Reserved



The floating-point operand in register **frA** is divided by the floating-point operand in register **frB**. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNaN, VXIDI, VXZDZ

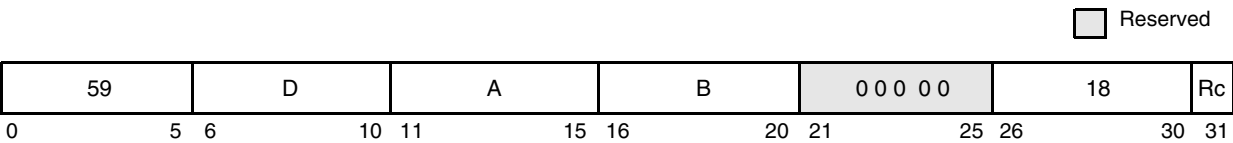
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fdivsx

Floating Divide Single (x'EC00 0024')

fdivsx

fdivs **frD,frA,frB** (Rc = '0')
fdivs. **frD,frA,frB** (Rc = '1')



The floating-point operand in register **frA** is divided by the floating-point operand in register **frB**. The remainder is not supplied as a result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero-divide exceptions when FPSCR[ZE] = '1'.

If the HID2[PSE] = '1', the quotient is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNaN, VXIDI, VXZDZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

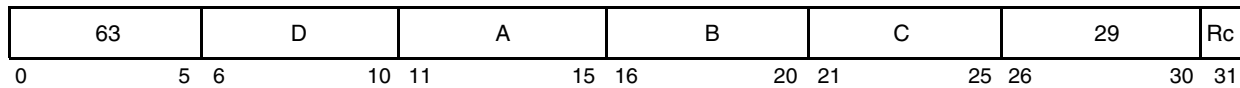
fmadd_x

Floating Multiply-Add (Double-Precision) (x'FC00 003A')

fmadd_x

fmadd **frD,frA,frC,frB** (Rc = '0')

fmadd. **frD,frA,frC,frB** (Rc = '1')



The following operation is performed:

$$\mathbf{frD} \leftarrow ((\mathbf{frA}) \times (\mathbf{frC})) + (\mathbf{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

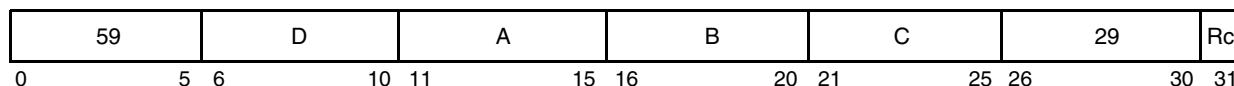
fmaddsx

Floating Multiply-Add Single (x'EC00 003A')

fmaddsx

fmaddsx **frD,frA,frC,frB** (Rc = '0')

fmaddsx. **frD,frA,frC,frB** (Rc = '1')



The followings operation are performed:

```

if HID2[PSE] = 0 then
    frD ← ((frA) × (frC)) + (frB)
else
    frD_ps0 ← ((frA_ps0) × (frC_ps0)) + (frB_ps0)
    frD_ps1 ← ((frA_ps0) × (frC_ps0)) + (frB_ps0)

```

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If the HID2[PSE] = '1', the result is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

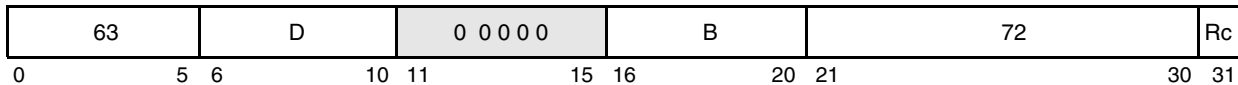
fmr_x

fmr_x

Floating Move Register (Double-Precision) (x'FC00 0090')

fmr **frD,frB** (Rc = '0')
fmr. **frD,frB** (Rc = '1')

 Reserved



The content of register **frB** is placed into **frD**.

If HID2[PSE] = '1' and the content in **frB** is a double-precision floating-point operand, the operand is copied to **frD**.

If HID2[PSE] = '1' and the content of **frB** contains a paired-single floating-point operand, the **frB**[ps0] is copied to **frD**[ps0] and the content of **frD**[ps1] is unchanged.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

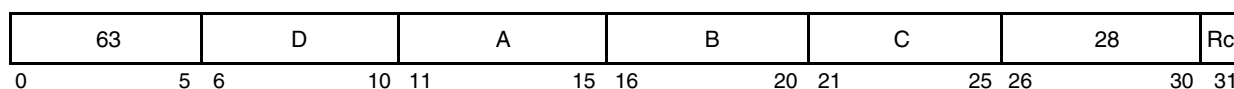
fmsub_x

fmsub_x

Floating Multiply-Subtract (Double-Precision) (x'FC00 0038')

fmsub **frD,frA,frC,frB** (Rc = '0')

fmsub. **frD,frA,frC,frB** (Rc = '1')



The following operation is performed:

$$\mathbf{frD} \leftarrow ((\mathbf{frA}) \times (\mathbf{frC})) - (\mathbf{frB})$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fmsubsx

Floating Multiply-Subtract Single (x'EC00 0038')

fmsubsx

fmsubs **frD,frA,frC,frB** (Rc = '0')
fmsubs. **frD,frA,frC,frB** (Rc = '1')

59	D	A	B	C	28	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← ((frA) × (frC)) - (frB)
else
    frD_ps0 ← ((frA_ps0) × (frC_ps0)) - (frB_ps0)
    frD_ps1 ← ((frA_ps0) × (frC_ps0)) - (frB_ps0)

```

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If HID2[PSE] = '1', the result is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

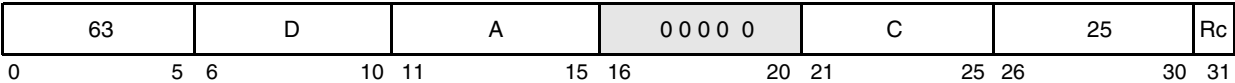
fmul_x

fmul_x

Floating Multiply (Double-Precision) (x'FC00 0032')

fmul **frD,frA,frC** (Rc = '0')
fmul. **frD,frA,frC** (Rc = '1')

 Reserved



The following operation is performed:

frD ← (**frA**) × (**frC**)

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

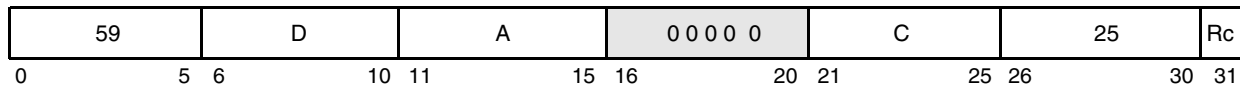
fmuls_x

Floating Multiply Single (x'EC00 0032')

fmuls_x

fmuls **frD,frA,frC** (Rc = '0')
fmuls. **frD,frA,frC** (Rc = '1')

 Reserved



The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← (frA) × (frC)
else
    frD_ps0 ← (frA_ps0) × (frC_ps0)
    frD_ps1 ← (frA_ps0) × (frC_ps0)
    
```

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If HID2[PSE] = '1', the result is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fnabs_x

Floating Negative Absolute Value (x'FC00 0110')

fnabs_x

fnabs

frD,frB

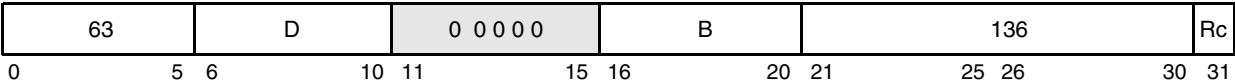
(Rc = '0')

fnabs.

frD,frB

(Rc = '1')

Reserved



The following operation is performed:

frD

←

1

||

frB[1-63]

The contents of register **frB** with bit 0 set are placed into **frD**.

Note: The **fnabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN can be altered by **fnabs**.

This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field)

Affected: FX, FEX, VX, OX

(if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

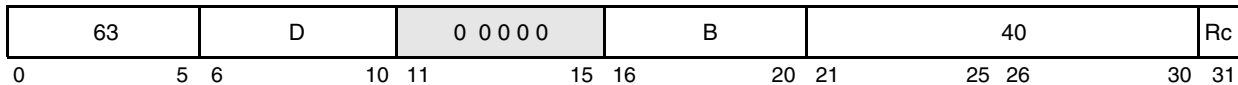
fneg_x

Floating Negate (x'FC00 0050')

fneg_x

fneg frD,frB (Rc = '0')

fneg. frD,frB (Rc = '1')

☐ Reserved


The following operation is performed:

$$\mathbf{frD} \leftarrow \neg \mathbf{frB}[0] \mid \mid \mathbf{frB}[1-63]$$

The contents of register **frB** with bit 0 inverted are placed into **frD**.

Note: The **fneg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN can be altered by **fneg**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

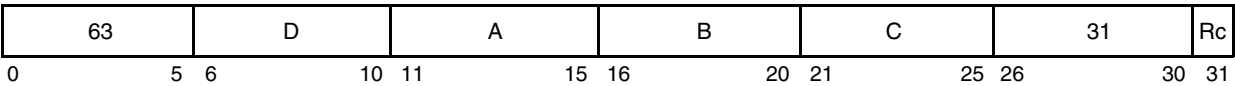
fnmadd_x

fnmadd_x

Floating Negative Multiply-Add (Double-Precision) (x'FC00 003E')

fnmadd **frD,frA,frC,frB** (Rc = '0')

fnmadd. **frD,frA,frC,frB** (Rc = '1')



The following operation is performed:

$$\mathbf{frD} \leftarrow - \left(\left(\mathbf{frA} \right) \times \left(\mathbf{frC} \right) \right) + \left(\mathbf{frB} \right)$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add (**fmadd_x**) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

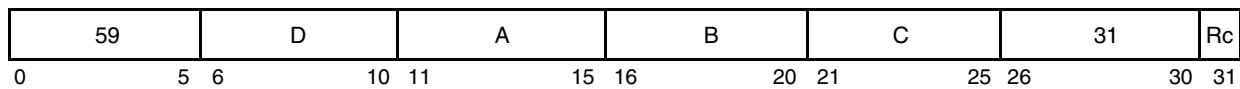
fnmadds_x

Floating Negative Multiply-Add Single (x'EC00 003E')

fnmadds_x

fnmadds **frD,frA,frC,frB** (Rc = '0')

fnmadds. **frD,frA,frC,frB** (Rc = '1')



The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← -(((frA) × (frC)) + (frB))
else
    frD_ps0 ← -(((frA_ps0) × (frC_ps0)) + (frB_ps0))
    frD_ps1 ← -(((frA_ps0) × (frC_ps0)) + (frB_ps0))

```

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is added to this intermediate result. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add Single (**fmadds_x**) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If the HID2[PSE] = '1' then the result is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

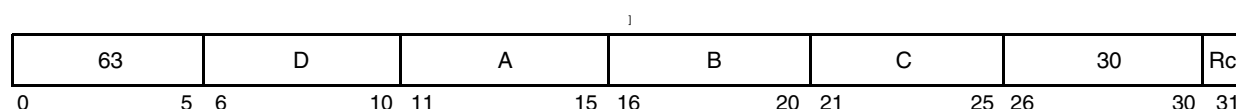
fnmsub_x

fnmsub_x

Floating Negative Multiply-Subtract (Double-Precision) (x'FC00 003C')

fnmsub **frD,frA,frC,frB** (Rc = '0')

fnmsub. **frD,frA,frC,frB** (Rc = '1')



The following operation is performed:

$$\mathbf{frD} \leftarrow - \left(((\mathbf{frA}) \times (\mathbf{frC})) - (\mathbf{frB}) \right)$$

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**fmsub_x**) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

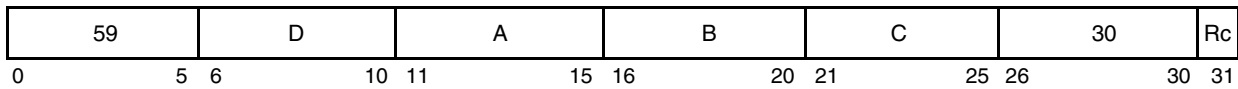
fnmsubs_x

Floating Negative Multiply-Subtract Single (x'EC00 003C')

fnmsubs_x

fnmsubs **frD,frA,frC,frB** (Rc = '0')

fnmsubs. **frD,frA,frC,frB** (Rc = '1')



The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← -(((frA) × (frC)) - (frB))
else
    frD_ps0 ← -(((frA_ps0) × (frC_ps0)) - (frB_ps0))
    frD_ps1 ← -(((frA_ps0) × (frC_ps0)) - (frB_ps0))

```

The floating-point operand in register **frA** is multiplied by the floating-point operand in register **frC**. The floating-point operand in register **frB** is subtracted from this intermediate result.

If the most-significant bit of the resultant significand is not '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **frD**.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract Single (**fmsubs_x**) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If HID2[PSE] = '1', the result is placed in both **frD_ps0** and **frD_ps1**.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fres_x

Floating Reciprocal Estimate Single (x'EC00 0030')

fres_x

fres **frD,frB** (Rc = '0')

fres. **frD,frB** (Rc = '1')

 Reserved


```

if HID2[PSE] = 0 then
    frD ← estimate[ 1/(frB) ]
else
    frD_ps0 ← estimate[ 1/(frB_ps0) ]
    frD_ps1 ← (frD_ps0)

```

A single-precision estimate of the reciprocal of the floating-point operand in register **frB** is placed into register **frD**. The estimate placed into register **frD** is correct to a precision of one part in 4096 of the reciprocal of **frB**. That is,

$$ABS \left(\frac{estimate - \left(\frac{1}{x}\right)}{\left(\frac{1}{x}\right)} \right) \leq \frac{1}{(4096)}$$

where *x* is the initial value in **frB**. Note that the value placed into register **frD** can vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in *Table 10-9*.

Table 10-9. Operand Values (fres_x)

Operand	Result	Exception
− <i>x</i>	−0	None
−0	− <i>x</i> ¹	ZX
+0	+ <i>x</i> ¹	ZX
+ <i>x</i>	+0	None
SNaN	QNaN ²	VXSNAN
QNaN	QNaN	None

1. No result if FPSCR[ZE] = '1'.
2. No result if FPSCR[VE] = '1'.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Note: The PowerPC Architecture makes no provision for a double-precision version of the **fresx** instruction. This is because graphics applications are expected to require only the single-precision version. No other important performance-critical applications are expected to require a double-precision version of the **fresx** instruction.

If HID2[PSE] = '1', the result is placed in both **frD_ps0** and **frD_ps1**.

This instruction is optional in the PowerPC Architecture. Other registers altered:

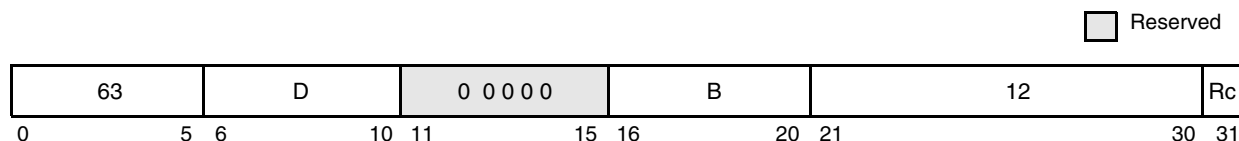
- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNaN

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA			YES	A

frsp_x

Floating Round to Single (x'FC00 0018')

frsp	frD,frB	(Rc = '0')
frsp.	frD,frB	(Rc = '1')

frsp_x

The following operation is performed:

```

if HID2[PSE] = 0 then
    frD ← Round Single( frB )
else
    frD_ps0 ← Round Single( frB_ps0 )
    frD_ps1 ← undefined

```

If `HID2[PSE] = '0'`, the floating-point operand in register `frB` is rounded to single-precision using the rounding mode specified by `FPSCR[RN]` and placed into `frD`.

If **HID2[PSE]** = '1', the source operand in register **frB** is rounded to single-precision using the rounding mode specified by **FPSCR[RN]** and placed into **frD_ps0**. The value in **frD_ps1** is undefined.

The rounding is described in the floating-point round to single-precision model section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNA

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

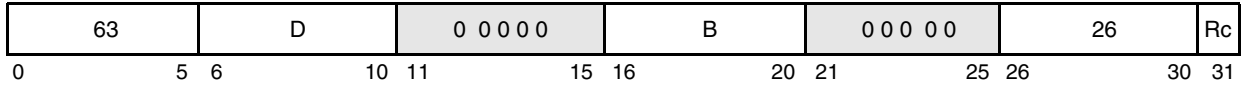
frsqrtex

Floating Reciprocal Square Root Estimate (x'FC00 0034')

frsqrtex

frsqrtex **frD,frB** (Rc = '0')
frsqrtex. **frD,frB** (Rc = '1')

☐ Reserved



A double-precision estimate of the reciprocal of the square root of the floating-point operand in register **frB** is placed into register **frD**. The estimate placed into register **frD** is correct to a precision of one part in 4096 of the reciprocal of the square root of **frB**. That is,

$$ABS \left(\frac{estimate - \left(\frac{1}{\sqrt{x}} \right)}{\left(\frac{1}{\sqrt{x}} \right)} \right) \leq \frac{1}{4096}$$

where x is the initial value in **frB**.

Note: The value placed into register **frD** can vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized in *Table 10-10*:

Table 10-10. Operand Values (frsqrtex)

Operand	Result	Exception
−x	QNaN ²	VXSQRT
<0	QNaN ²	VXSQRT
−0	−x ¹	ZX
+0	+x ¹	ZX
+x	+0	None
SNaN	QNaN ²	VXSNAN
QNaN	QNaN	None

1. No result if FPSCR[ZE] = '1'.

2. No result if FPSCR[VE] = '1'.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Note: No single-precision version of the **frsqrtex** instruction is provided; however, if **frB** is representable in single-precision format, then **frD** will also be representable in single-precision format.

This instruction is optional in the PowerPC Architecture.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR (undefined), FI (undefined), FX, ZX, VXSNaN, VXSQRT

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA			Yes	A

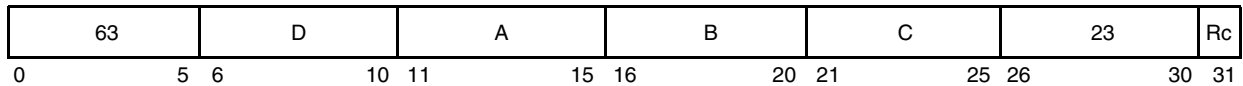
fsel_x

Floating Select (x'FC00 002E')

fsel_x

fsel **frD,frA,frC,frB** (Rc = '0')

fsel. **frD,frA,frC,frB** (Rc = '1')



```

if (frA) ≥ 0.0 then
    frD ← (frC)
else
    frD ← (frB)

```

The floating-point operand in register **frA** is compared to the value zero. If the operand is greater than or equal to zero, register **frD** is set to the contents of register **frC**. If the operand is less than zero or is a NaN, the register **frD** is set to the contents of register **frB**. The comparison ignores the sign of zero (that is, regards +0 as equal to -0).

Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

For examples of uses of this instruction, see the floating-point conversions section and the floating-point selection section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

This instruction is optional in the PowerPC Architecture.

If HID2[PSE] = '1' and the selected source is a double-precision floating-point operand, the selected operand from **frB** or **frC** is copied to **frD** (as described previously).

If HID2[PSE] = '1' and the selected source contains paired-single floating-point operands, only **frA_ps0** is compared to zero, and the selected operand from **frB_ps0** or **frC_ps0** is copied to **frD_ps0**. The content of **frD_ps1** is undefined.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA			Yes	A

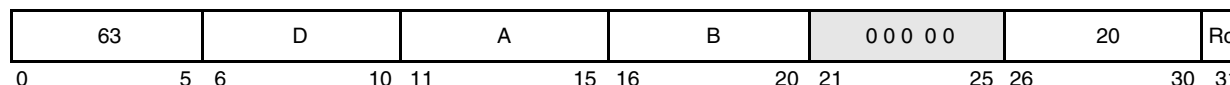
fsub_x

fsub_x

Floating Subtract (Double-Precision) (x'FC00 0028')

fsub **frD,frA,frB** (Rc = '0')

fsub. **frD,frA,frB** (Rc = '1')

 Reserved


The following operation is performed:

$$\mathbf{frD} \leftarrow (\mathbf{frA}) - (\mathbf{frB})$$

The floating-point operand in register **frB** is subtracted from the floating-point operand in register **frA**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD**.

The execution of the **fsub** instruction is identical to that of **fadd**, except that the contents of **frB** participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered:

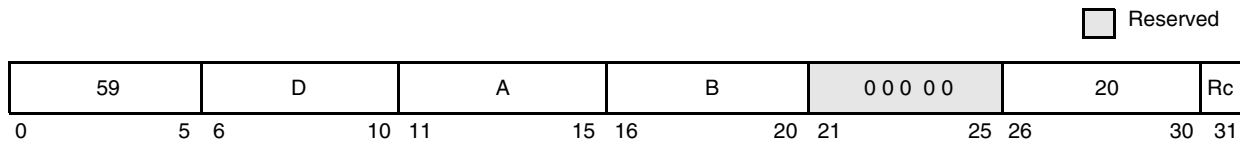
- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

fsubsx

Floating Subtract Single (x'EC00 0028')

fsubs	frD,frA,frB	(Rc = '0')
fsubs.	frD,frA,frB	(Rc = '1')

fsubs_x

The following operations are performed:

```

if HID2[PSE] = 0 then
    frD ← (frA) - (frB)
else
    frD_ps0 ← (frA_ps0) - (frB_ps0)
    frD_ps1 ← (frA_ps0) - (frB_ps0)

```

The floating-point operand in register **frB** is subtracted from the floating-point operand in register **frA**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field **RN** of the **FPSCR** and placed into **frD**.

The execution of the **fsubs** instruction is identical to that of **fadds**, except that the contents of **frB** participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = '1'.

If `HID2[PSE] = '1'`, the result is placed in both `frD_ps0` and `frD_ps1`.

Other registers altered:

- **Condition Register (CR1 field)**
Affected: FX, FEX, VX, OX (if Rc = '1')
- **Floating-Point Status and Control Register**
Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

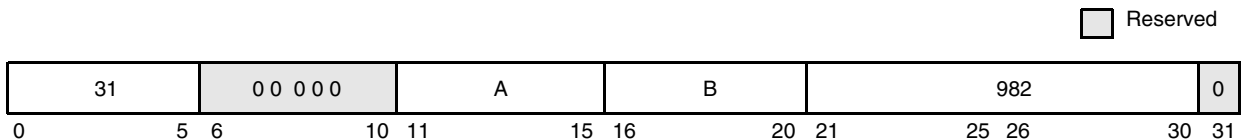
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				A

icbi

Instruction Cache Block Invalidate (x'7C00 07AC')

icbi

icbi **rA,rB**



EA is the sum (rA10) + (rB).

If the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, then the block is made invalid in all such instruction caches, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, then the block is made invalid in that instruction cache, so that subsequent references cause the block to be refetched.

The function of this instruction is independent of the write-through, write-back, and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for referenced and changed bit recording except that referenced and changed bit recording might not occur. This instruction has no effect on the unified L2 cache.

The **icbi** instruction invalidates the block at EA (rA10 + rB). If the block is marked coherency-required, the processor sends an address-only broadcast to other processors causing those processors to invalidate the block from their instruction caches.

For faster processing, Espresso does not compare the entire EA (rA10 + rB) with the tag in the instruction cache. Instead, it uses the bits in the EA to locate the set that the block is in, and invalidates all blocks in that set.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				X

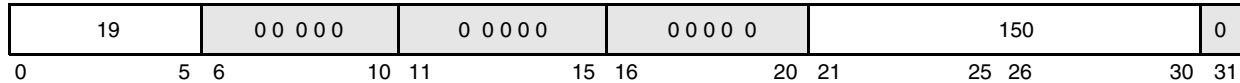
isync

Instruction Synchronize (x'4C00 012C')

isync

isync

Reserved



The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that memory accesses caused by those instructions are not required to have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the **isync** instruction. The **isync** instruction has no effect on the other processors or on their caches.

This instruction is context synchronizing.

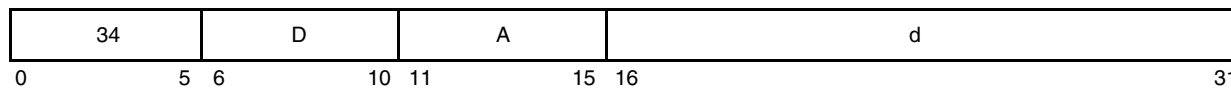
Context synchronization is necessary after certain code sequences that perform complex operations within the processor. These code sequences are typically operating system tasks that involve memory management. For example, if an instruction A changes the memory translation rules in the memory management unit (MMU), the **isync** instruction should be executed so that the instructions following instruction A will be discarded from the pipeline and refetched according to the new translation rules.

Note: All exceptions and the **rfi** and **sc** instructions are also context synchronizing.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				XL

lbz**lbz** **rD,d(rA)**

```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
rD ← (24)0 || MEM(EA, 1)

```

Other registers altered:

- None

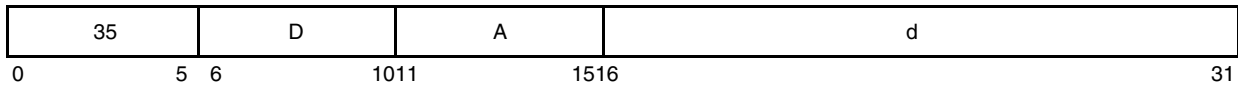
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lbzu

lbzu

Load Byte and Zero with Update (x'8C00 0000')

lbzu **rD,d(rA)**



$$EA \leftarrow (rA) + EXTS(d)$$

$$rD \leftarrow (24)0 \parallel MEM(EA, 1)$$

$$rA \leftarrow EA$$

EA is the sum (rA) + d. The byte in memory addressed by EA is loaded into the low-order 8 bits of rD. The remaining bits in rD are cleared.

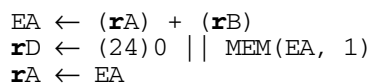
EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lbzux**lbzux** **rD,rA,rB**

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lbzx

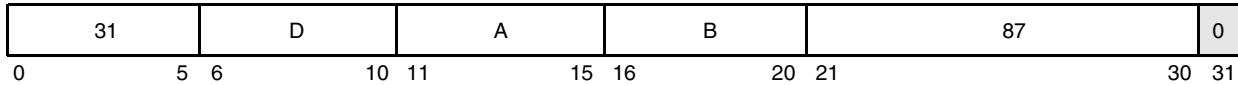
Load Byte and Zero Indexed (x'7C00 00AE')

lbzx

lbzx

rD,rA,rB

☐ Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
rD ← (24)0 || MEM(EA, 1)
    
```

EA is the sum (rA|0) + (rB). The byte in memory addressed by EA is loaded into the low-order 8 bits of rD. The remaining bits in rD are cleared.

Other registers altered:

- None

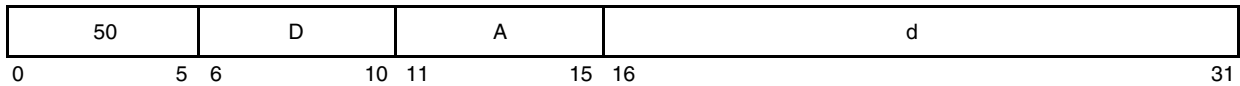
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lfd

lfd

Load Floating-Point Double (x'C800 0000')

lfd **frD,d(rA)**



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
frD ← MEM(EA, 8)
```

EA is the sum (**rA**l0) + d.

The doubleword in memory addressed by EA is placed into **frD**.

Other registers altered:

- None

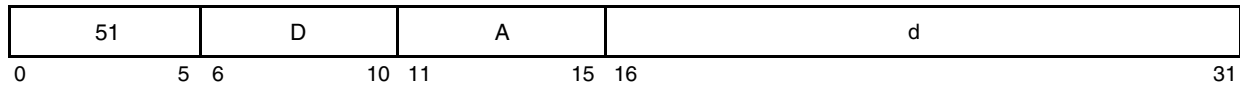
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lfd

lfd

Load Floating-Point Double with Update (x'CC00 0000')

lfd **frD,d(rA)**



```
EA ← (rA) + EXTS(d)
frD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (rA) + d.

The doubleword in memory addressed by EA is placed into **frD**.

EA is placed into **rA**.

If **rA** = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

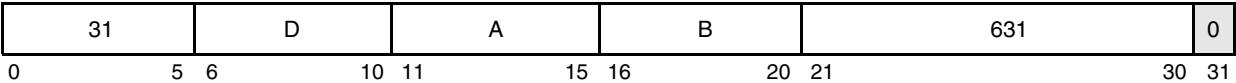
lfdux

lfdux

Load Floating-Point Double with Update Indexed (x'7C00 04EE')

lfdux **frD,rA,rB**

 Reserved



$EA \leftarrow (rA) + (rB)$
 $frD \leftarrow MEM(EA, 8)$
 $rA \leftarrow EA$

EA is the sum (rA) + (rB).
The doubleword in memory addressed by EA is placed into frD.
EA is placed into rA.
If rA = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

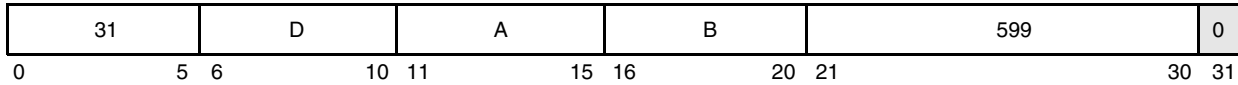
lfdx

lfdx

Load Floating-Point Double Indexed (x'7C00 04AE')

lfdx **frD,rA,rB**

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
frD ← MEM(EA, 8)
    
```

EA is the sum (**rA**l0) + (**rB**).

The doubleword in memory addressed by EA is placed into **frD**.

Other registers altered:

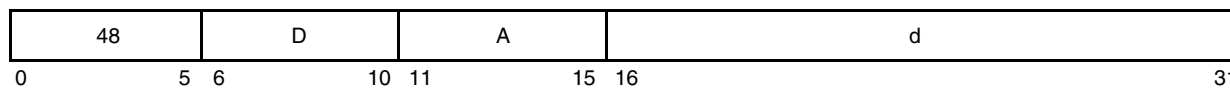
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lfs

lfs

Load Floating-Point Single (x'C000 0000')

lfs **frD,d(rA)**

```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
if HID2[PSE] = 0 then
    frD ← DOUBLE(MEM(EA, 4))
else
    frD_ps0 ← Single(MEM(EA, 4))
    frD_ps1 ← Single(MEM(EA, 4))

```

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = '0', this word is converted to floating-point double-precision and placed into **frD**.

If HID2[PSE] = '1', this word is interpreted as a floating-point single-precision operand and placed into **frD_ps0** and replicated in **frD_ps1**.

Other registers altered:

- None

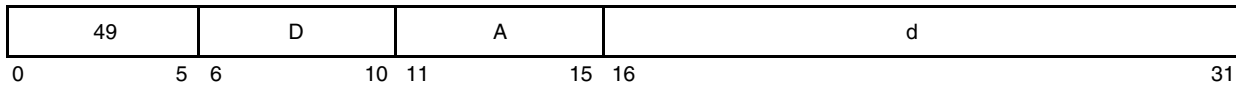
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lfsu

lfsu

Load Floating-Point Single with Update (x'C400 0000')

lfsu **frD,d(rA)**



```

EA ← (rA) + EXTS(d)
rA ← EA
if HID2[PSE] = 0 then
    frD ← DOUBLE(MEM(EA, 4))
else
    frD_ps0 ← Single(MEM(EA, 4))
    frD_ps1 ← Single(MEM(EA, 4))

```

EA is the sum (rA) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = '0', this word is converted to floating-point double-precision and placed into frD.

If HID2[PSE] = '1', this word is interpreted as a floating-point single-precision operand and placed into frD_ps0 and replicated in frD_ps1.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

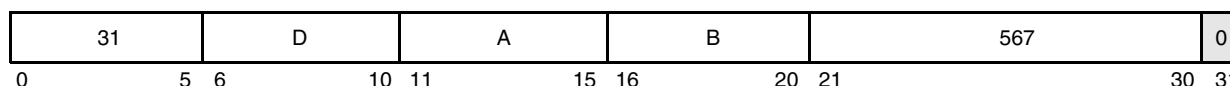
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lfsux

lfsux

Load Floating-Point Single with Update Indexed (x'7C00 046E')

lfsux**frD,rA,rB** Reserved

```

EA ← (rA) + (rB)
if HID2[PSE] = 0 then
    frD ← DOUBLE(MEM(EA, 4))
else
    frD_ps0 ← Single(MEM(EA, 4))
    frD_ps1 ← Single(MEM(EA, 4))
rA ← EA

```

EA is the sum (rA) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = '0', this word is converted to floating-point double-precision and placed into **frD** (see the floating-point load instructions section of the *PowerPC Microprocessor Family: The Programming Environments manual*).

If HID2[PSE] = '1', this word is interpreted as a floating-point single-precision operand and placed into **frD_ps0** and replicated in **frD_ps1**.

EA is placed into **rA**.

If **rA** = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lfsx

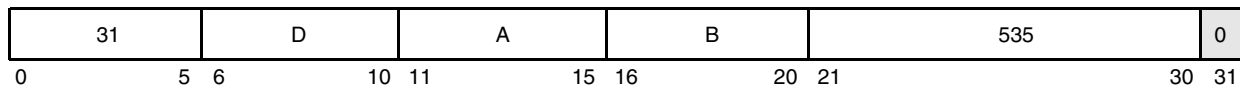
lfsx

Load Floating-Point Single Indexed (x'7C00 042E')

lfsx

frD,rA,rB

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
if HID2[PSE] = 0 then
    frD ← DOUBLE(MEM(EA, 4))
else
    frD_ps0 ← Single(MEM(EA, 4))
    frD_ps1 ← Single(MEM(EA, 4))

```

EA is the sum (rA|0) + (rB).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand.

If HID2[PSE] = '0', this word is converted to floating-point double-precision and placed into **frD**.

If HID2[PSE] = '1', this word is interpreted as a floating-point single-precision operand and placed into **frD_ps0** and replicated in **frD_ps1**.

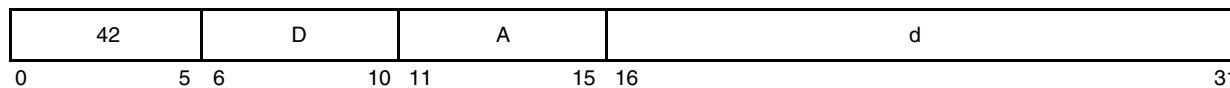
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				X

Iha

lha	rD,d(rA)
-----	----------



EA is the sum ($\mathbf{rA}[0] + d$). The halfword in memory addressed by EA is loaded into the low-order 16 bits of $\mathbf{rD}$. The remaining bits in $\mathbf{rD}$ are filled with a copy of the most-significant bit of the loaded halfword.

- None

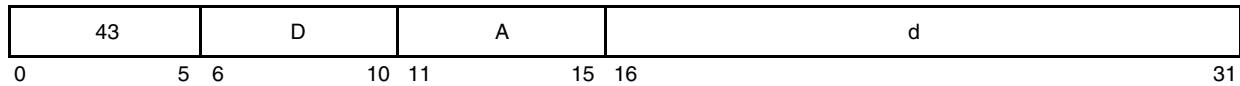
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lhau

lhau

Load Halfword Algebraic with Update (x'AC00 0000')

lhau rD,d(rA)



```
EA ← (rA) + EXTS(d)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (rA) + d. The halfword in memory addressed by EA is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most-significant bit of the loaded halfword.

EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

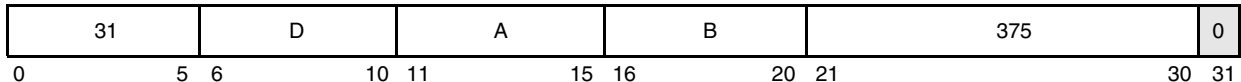
lhaux

lhaux

Load Halfword Algebraic with Update Indexed (x'7C00 02EE')

lhaux **rD,rA,rB**

 Reserved



$EA \leftarrow (rA) + (rB)$
 $rD \leftarrow EXTS(MEM(EA, 2))$
 $rA \leftarrow EA$

EA is the sum (rA) + (rB). The halfword in memory addressed by EA is loaded into the low-order 16 bits of rD. The remaining bits in rD are filled with a copy of the most-significant bit of the loaded halfword.

EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

Other registers altered:

- None

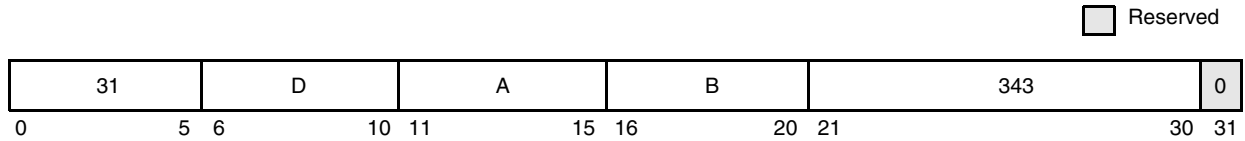
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lhax

lhax

Load Halfword Algebraic Indexed (x'7C00 02AE')

Ihax

 r_D, r_A, r_B 

```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
rD ← EXTS(MEM(EA, 2))

```

EA is the sum (**rA**10) + (**rB**). The halfword in memory addressed by EA is loaded into the low-order 16 bits of **rD**. The remaining bits in **rD** are filled with a copy of the most-significant bit of the loaded halfword.

Other registers altered:

- None

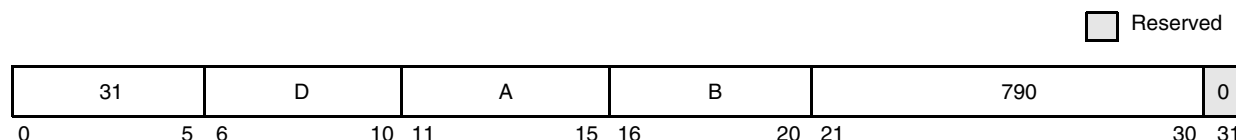
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

Ihbrx

lhbrx

Load Halfword Byte-Reverse Indexed (x'7C00 062C')

lhbrx rD,rA,rB



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA + 1, 1) || MEM(EA, 1)

```

EA is the sum (**rA**10) + (**rB**). Bits 0:7 of the halfword in memory addressed by EA are loaded into the low-order 8 bits of **rD**. Bits 8:15 of the halfword in memory addressed by EA are loaded into the subsequent low-order 8 bits of **rD**. The remaining bits in **rD** are cleared.

Other registers altered:

- None

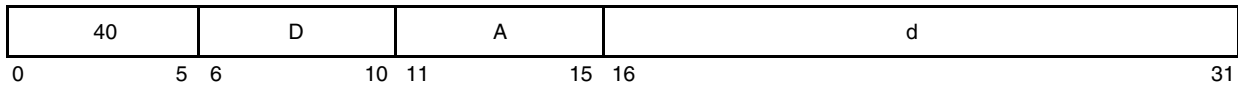
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lhz

lhz

Load Halfword and Zero (x'A000 0000')

lhz **rD,d(rA)**



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
    
```

EA is the sum (**rA**l0) + **d**. The halfword in memory addressed by EA is loaded into the low-order 16 bits of **rD**. The remaining bits in **rD** are cleared.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lhzux

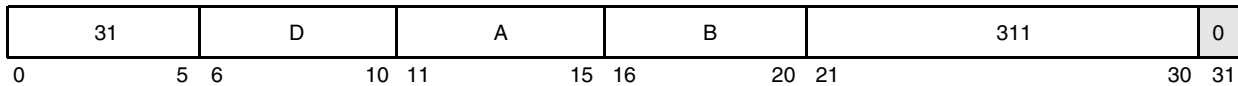
lhzux

Load Halfword and Zero with Update Indexed (x'7C00 026E')

lhzux

rD,rA,rB

 Reserved



$$EA \leftarrow (rA) + (rB)$$

$$rD \leftarrow (16)0 \parallel MEM(EA, 2)$$

$$rA \leftarrow EA$$

EA is the sum (rA) + (rB). The halfword in memory addressed by EA is loaded into the low-order 16 bits of rD. The remaining bits in rD are cleared.

EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

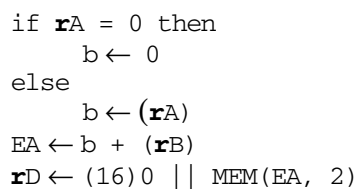
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lhzx

lhzx rD,rA,rB



Other registers altered:

- None

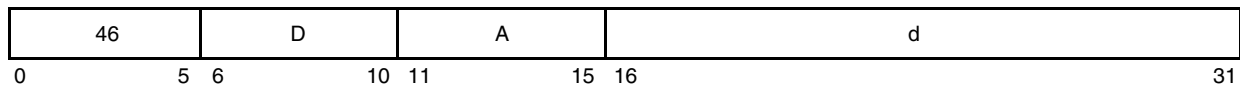
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

Imw

Load Multiple Word (x'B800 0000')

Imw

Imw **rD,d(rA)**



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
r ← rD
do while r ≤ 31
    GPR(r) ← MEM(EA, 4)
    r ← r + 1
    EA ← EA + 4

```

EA is the sum (rA10) + d.

$n = (32 - rD)$.

n consecutive words starting at EA are loaded into GPRs rD through r31.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see the DSI exception (x'00300') section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

If rA is in the range of registers specified to be loaded, including the case in which rA = 0, the instruction form is invalid.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

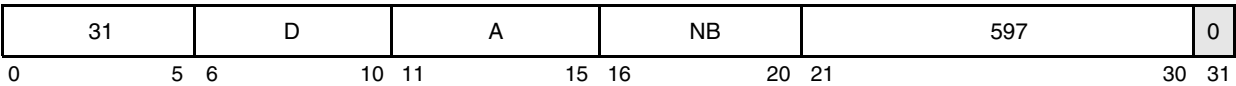
lswi

lswi

Load String Word Immediate (x'7C00 04AA')

lswi **rD,rA,NB**

Reserved



```

if rA = 0 then
    EA ← 0
else
    EA ← (rA)
if NB = 0 then
    n ← 32
else
    n ← NB
r ← rD - 1
i ← 0
do while n > 0
    if i = 0 then
        r ← r + 1 (mod 32)
        GPR(r) ← 0
    GPR(r) [i, i + 7] ← MEM(EA, 1)
    i ← i + 8
    if i = 32 then
        i ← 0
    EA ← EA + 1
    n ← n - 1
    
```

EA is (rA|0). Let $n = NB$ if $NB \neq 0$, $n = 32$ if $NB = 0$; n is the number of bytes to load.
 Let $nr = \text{CEIL}(n \div 4)$; nr is the number of registers to be loaded with data.
 n consecutive bytes starting at EA are loaded into GPRs rD through rD + nr - 1.

Bytes are loaded left to right in each register. The sequence of registers wraps around to r0 if required. If the low-order 4 bytes of register rD + nr - 1 are only partially filled, the unfilled low-order bytes of that register are cleared.

If rA is in the range of registers specified to be loaded, including the case in which rA = 0, the instruction form is invalid. Under certain conditions (for example, segment boundary crossing) the data alignment exception handler might be invoked. For additional information about data alignment exceptions, see the DSI exception (x'00300') section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In some implementations, this instruction is likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

- Other registers altered:
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

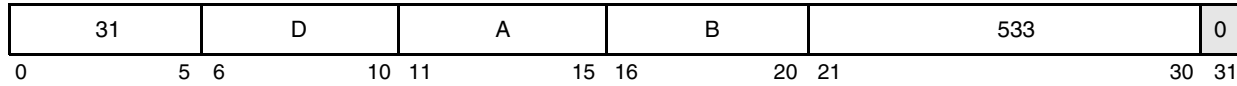
lswx

Load String Word Indexed (x'7C00 042A')

lswx

lswx

rD,rA,rB

 Reserved


```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
n ← XER[25:31]
r ← rD - 1
i ← 0
rD ← undefined
do while n > 0
    if i = 0 then
        r ← r + 1 (mod 32)
        GPR(r) ← (32)0
        GPR(r)[i,i + 7] ← MEM(EA, 1)
        i ← i + 8
        if i = 32 then
            i ← 0
        EA ← EA + 1
        n ← n - 1

```

EA is the sum (rA10) + (rB). Let $n = \text{XER}[25:31]$; n is the number of bytes to load. Let $nr = \text{CEIL}(n \div 4)$; nr is the number of registers to receive data. If $n > 0$, n consecutive bytes starting at EA are loaded into GPRs rD through rD + nr - 1. Bytes are loaded left to right in each register. The sequence of registers wraps around through r0 if required. If the 4 bytes of rD + nr - 1 are only partially filled, the unfilled low-order bytes of that register are cleared. If $n = 0$, the contents of rD are undefined.

If rA or rB is in the range of registers specified to be loaded, including the case in which rA = 0, either the system illegal instruction error handler is invoked or the results are boundedly undefined. If rD = rA or rD = rB, the instruction form is invalid; If rD and rA both specify GPR0, the form is invalid.

Under certain conditions (for example, segment boundary crossing), the data alignment exception handler may be invoked. For additional information about data alignment exceptions, see the DSI exception (x'00300') section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

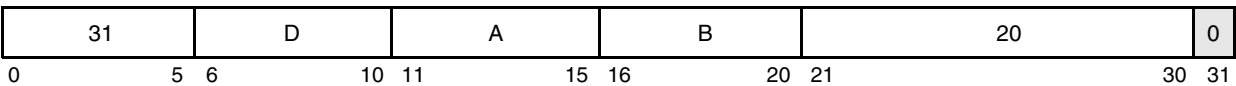
lwarx

lwarx

Load Word and Reserve Indexed (x'7C00 0028')

lwarx **rD,rA,rB**

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← MEM(EA, 4)
    
```

EA is the sum (**rA**l0) + (**rB**).

The word in memory addressed by **EA** is loaded into **rD**.

This instruction creates a reservation for use by a store word conditional indexed (**stwcx.**)instruction. The physical address computed from **EA** is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see the DSI exception (x'00300') section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

When the **RESERVE** bit is set, the processor enables hardware snooping for the block of memory addressed by the **RESERVE** address. If the processor detects that another processor writes to the block of memory it has reserved, it clears the **RESERVE** bit. The **stwcx.** instruction performs store operations only if the **RESERVE** bit is set. The **stwcx.** instruction sets the **CR0[EQ]** bit if the store was successful and clears it if it failed. The **lwarx** and **stwcx.** combination can be used for atomic read-modify-write sequences.

Note: The atomic sequence is not guaranteed, but its failure can be detected if **CR0[EQ]** = '0' after the **stwcx.** instruction.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UIA				X

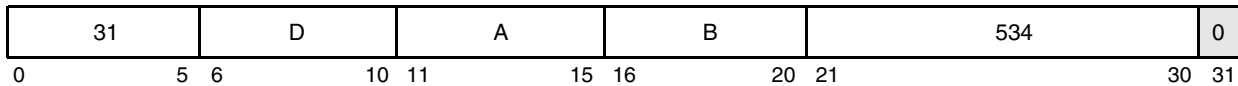
lwbrr

Load Word Byte-Reverse Indexed (x'7C00 042C')

lwbrr

lwbrr **rD,rA,rB**

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
rD ← MEM(EA + 3, 1) || MEM(EA + 2, 1) || MEM(EA + 1, 1) || MEM(EA, 1)

```

EA is the sum (rA|0) + rB. Bits 0:7 of the word in memory addressed by EA are loaded into the low-order 8 bits of rD. Bits 8:15 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of rD. Bits 16:23 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of rD. Bits 24:31 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of rD.

Other registers altered:

- None

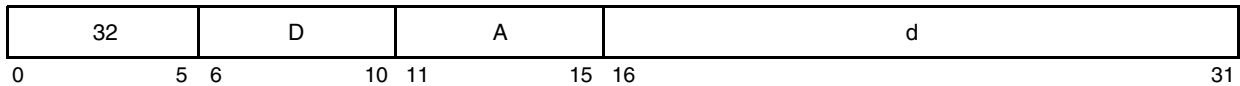
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

lwz

Load Word and Zero (x'8000 0000')

lwz

lwz **rD,d(rA)**



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
rD ← MEM(EA, 4)
```

EA is the sum (**rA**l0) + d. The word in memory addressed by EA is loaded into **rD**.

Other registers altered:

- None

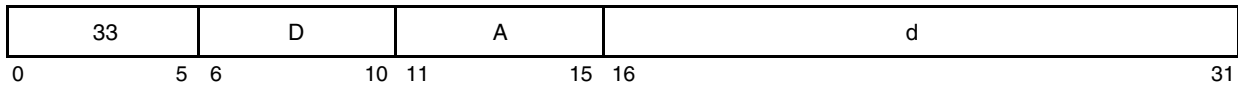
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

lwzu

lwzu

Load Word and Zero with Update (x'8400 0000')

lwzu rD,d(rA)



$$EA \leftarrow rA + EXTS(d)$$

$$rD \leftarrow MEM(EA, 4)$$

$$rA \leftarrow EA$$

EA is the sum (rA) + d. The word in memory addressed by EA is loaded into rD.

EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

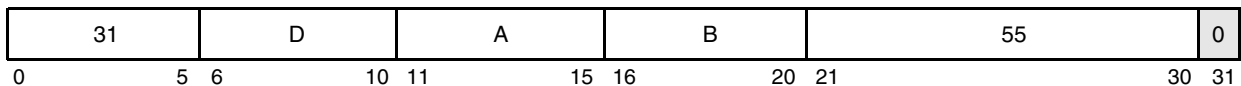
lwzux

lwzux

Load Word and Zero with Update Indexed (x'7C00 006E')

lwzux rD,rA,rB

 Reserved



$EA \leftarrow (rA) + (rB)$
 $rD \leftarrow MEM(EA, 4)$
 $rA \leftarrow EA$

EA is the sum (rA) + (rB). The word in memory addressed by EA is loaded into rD.

EA is placed into rA.

If rA = 0 or rA = rD, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

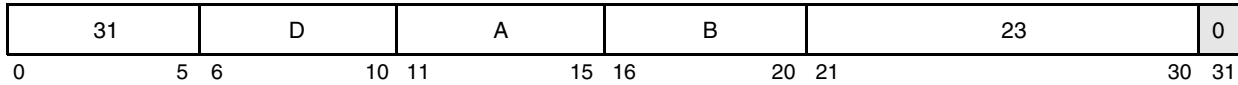
lwzx

lwzx

Load Word and Zero Indexed (x'7C00 002E')

lwzx **rD,rA,rB**

Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + rB
rD ← MEM(EA, 4)
    
```

EA is the sum (**rA**l0) + (**rB**). The word in memory addressed by EA is loaded into **rD**.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mcrfs

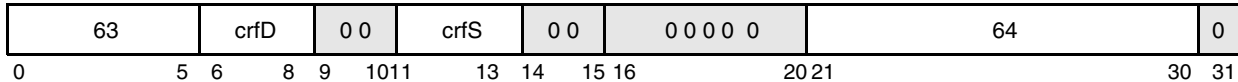
mcrfs

Move to Condition Register from FPSCR (x'FC00 0080')

mcrfs

crfD,crfS

 Reserved



The contents of FPSCR field **crfS** are copied to CR field **crfD**. All exception bits copied (except FEX and VX) are cleared in the FPSCR.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: FX, FEX, VX, OX
- Floating-Point Status and Control Register
Affected: FX, OX (if **crfS** = 0)
Affected: UX, ZX, XX, VXSNaN (if **crfS** = 1)
Affected: VXISI, VXIDI, VXZDZ, VXIMZ (if **crfS** = 2)
Affected: VXVC (if **crfS** = 3)
Affected: VXSOFT, VXSQRT, VXCVI (if **crfS** = 5)

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mcrxr

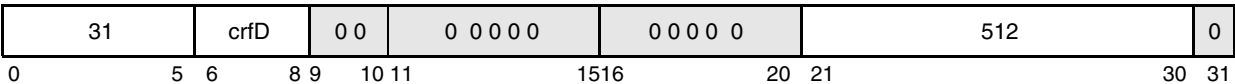
mcrxr

Move to Condition Register from XER (x'7C00 0400')

mcrxr

crfD

Reserved



$CR((4 \times \text{crfD}) - (4 \times \text{crfD} + 3)) \leftarrow XER[0-3]$
 $XER[0-3] \leftarrow 0b0000$

The contents of XER[0:3] are copied into the condition register field designated by **crfD**. All other fields of the condition register remain unchanged. XER[0:3] is cleared.

Other registers altered:

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, SO
- XER[0:3]

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mfcrr

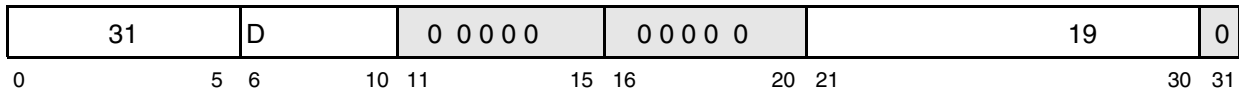
Move from Condition Register (x'7C00 0026')

mfcrr

mfcrr

rD

 Reserved



$rD \leftarrow CR$

The contents of the Condition Register (CR) are placed into rD.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

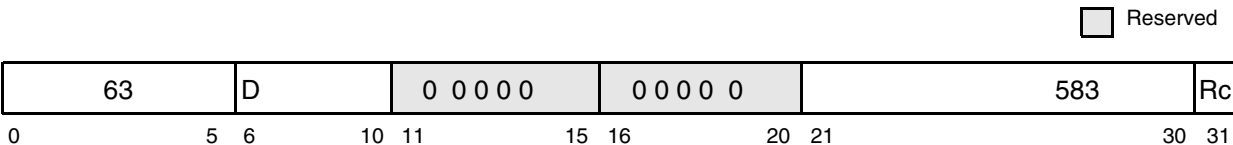
mffs_x

mffs_x

Move from FPSCR (x'FC00 048E')

mffsfrD(Rc = '0')

mffs.frD(Rc = '1')



frD[32-63] ← FPSCR

The contents of the Floating-Point Status and Control Register (FPSCR) are placed into the low-order bits of register **frD**. The high-order bits of register **frD** are undefined.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mfmsr

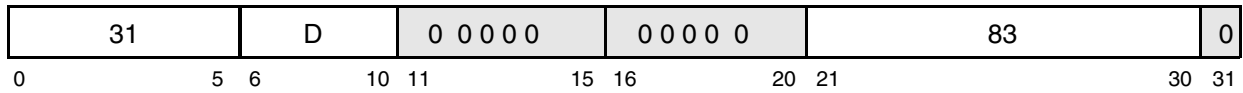
Move from Machine State Register (x'7C00 00A6')

mfmsr

mfmsr

rD

 Reserved



rD ← MSR

The contents of the MSR are placed into **rD**.

This is a supervisor-level instruction.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

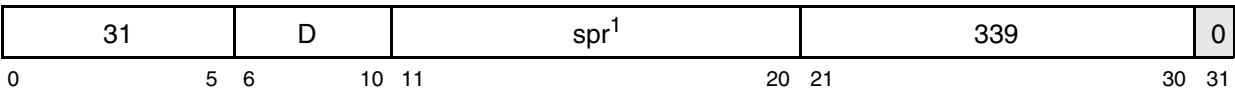
mfspr

mfspr

Move from Special-Purpose Register (x'7C00 02A6')

mfspr rD,SPR

Reserved



1. spr is a split field.

$n \leftarrow \text{spr}[5-9] \parallel \text{spr}[0-4]$
 $rD \leftarrow \text{SPR}(n)$

The SPR field denotes a special-purpose register. See *Table 2-36. SPR Encodings for PowerPC-Defined Registers* on page 99 and *Table 2-37. SPR Encodings for Espresso-Defined Registers* on page 101 for the spr encodings. The contents of the designated special purpose registers are placed into rD.

The **mfspr** instruction is supervisor-level only if SPR[0] = '1'. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = '1' results in a privileged instruction type program exception.

If MSR[PR] = '1', the only effect of executing an instruction with an SPR number that is not shown in *Table 2-36* on page 99 or *Table 2-37* on page 101 and has SPR[0] = '1' is to cause a supervisor-level instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = '0' or SPR[0] = '0'. If the SPR field contains any value that is not shown in *Table 2-36* on page 99 or *Table 2-37* on page 101, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- None

Simplified mnemonics:

mfxr	rD	equivalent to	mfspr	rD,1
mflr	rD	equivalent to	mfspr	rD,8
mfctr	rD	equivalent to	mfspr	rD,9

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA/OEA	Yes ¹			XFX
1. mfspr is supervisor-level only if SPR[0] = '1'.				

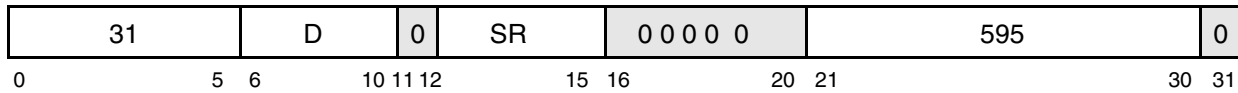
mfsr

Move from Segment Register (x'7C00 04A6')

mfsr

mfsr

rD,SR

 Reserved


$$rD \leftarrow \text{SEGREG}(\mathbf{SR})$$

The contents of the segment register SR are copied into rD.

This is a supervisor-level instruction.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

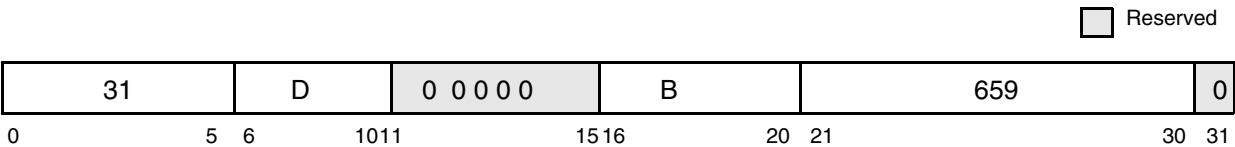


mfsrin

mfsrin

Move from Segment Register Indirect (x'7C00 0526')

mfsrin **rD,rB**



$rD \leftarrow \text{SEGREG}(rB[0-3])$

The contents of the segment register selected by bits 0:3 of **rB** are copied into **rD**.

This is a supervisor-level instruction.

The **rA** field is not defined for the **mfsrin** instruction in the PowerPC Architecture. However, **mfsrin** performs the same function in the PowerPC Architecture as does the **mfsri** instruction in the Power Architecture (if **rA** = '0').

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

mftb

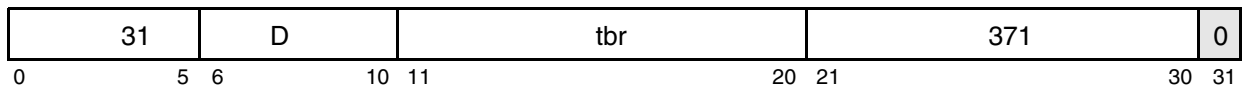
mftb

Move from Time Base (x'7C00 02E6')

mftb

rD,TBR

Reserved



Note: tbr is a split field.

```

n ← tbr[5-9] || tbr[0-4]
if n = 268 then
    rD ← TBL
else if n = 269 then
    rD ← TBU
else
    error(invalid TBR field)

```

The contents of TBL or TBU are copied into **rD**, as designated by the value in TBR, encoded as shown.

Table 10-11. TBR Encodings for *mftb*

TBR ¹			Register Name	Access
Decimal	tbr[5:9]	tbr[0:4]		
268	01000	01100	TBL	User
269	01000	01101	TBU	User

1. The order of the two 5-bit halves of the TBR number is reversed.

If the TBR field contains any value other than one of the values shown in *Table 10-11*, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined.

For more information about the time base, see the PowerPC VEA Register Set—Time Base section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

Simplified mnemonics:

mftb	rD	equivalent to	mftb	rD,268
mftbu	rD	equivalent to	mftb	rD,269

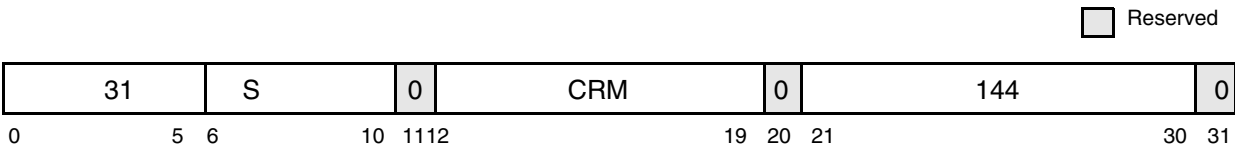
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
VEA				XFX

mtcrf

mtcrf

Move to Condition Register Fields (x'7C00 0120')

mtcrfCRM,rS



```
mask ← (4)(CRM[0]) || (4)(CRM[1]) || ... (4)(CRM[7])
CR ← (rS & mask) | (CR & ¬ mask)
```

The contents of rS are placed into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0 - 7. If CRM(i) = '1', CR field i (CR bits 4 × i through 4 × i + 3) is set to the contents of the corresponding field of rS.

Note: Updating a subset of the eight fields of the Condition Register might have substantially poorer performance on some implementations than updating all of the fields.

- Other registers altered:
- CR fields selected by mask

Simplified mnemonics:

mtcr rS equivalent to mtcrf 0xFF,rS

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XFX

mtfsb0_x

Move to FPSCR Bit 0 (x'FC00 008C')

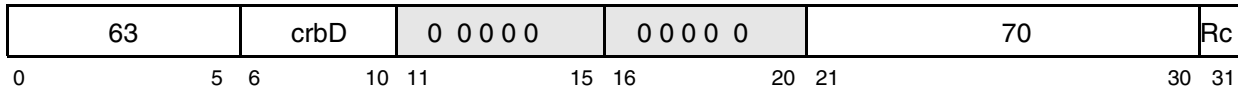
mtfsb0
mtfsb0.

crbD
crbD

(Rc = '0')
(Rc = '1')

mtfsb0_x

 Reserved



FPSCR (**crbD**) ← 0

Bit **crbD** of the FPSCR is cleared.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPSCR bit **crbD**

Note: Bit 1(FEX) and bit 2 (VX) cannot be explicitly cleared.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mtfsb1_x

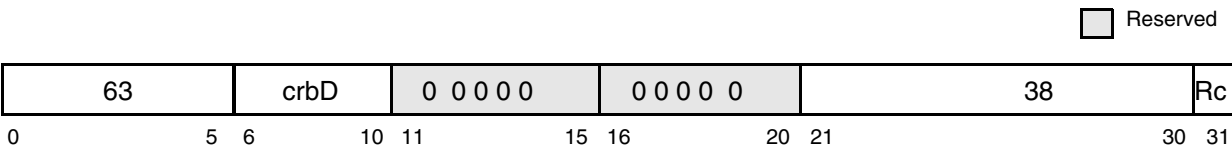
Move to FPSCR Bit 1 (x'FC00 004C')

mtfsb1_x

mtfsb1
mtfsb1.

crbD
crbD

(Rc = '0')
(Rc = '1')



FPSCR (**crbD**) ← 1

Bit **crbD** of the FPSCR is set.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPSCR bit **crbD** and FX

Note: Bit 1 (FEX) and bit 2 (VX) cannot be explicitly set.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

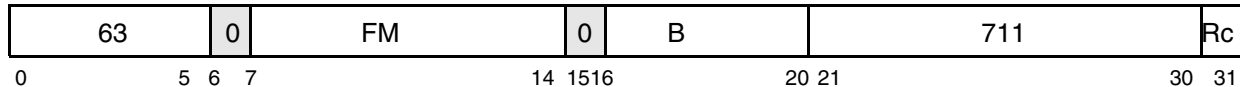
mtfsf_x

Move to FPSCR Fields (x'FC00 058E')

mtfsf_x

mtfsf FM,frB (Rc = '0')

mtfsf. FM,frB (Rc = '1')

☐ Reserved


The low-order 32 bits of **frB** are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0:7. If FM[*i*] = '1', FPSCR field *i* (FPSCR bits 4 × *i* through 4 × *i* + 3) is set to the contents of the corresponding field of the low-order 32 bits of register **frB**.

FPSCR[FX] is altered only if FM[0] = '1'.

Updating fewer than all eight fields of the FPSCR might have substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0:3] is specified, bits 0 (FX) and 3 (OX) are set to the values of **frB**[32] and **frB**[35] (that is, even if this instruction causes OX to change from '0' to '1', FX is set from **frB**[32] and not by the usual rule that FX is set when an exception bit changes from '0' to '1'). Bit 1 (FEX) and bit 2 (VX) are set according to the usual rule and not from **frB**[33:34].

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPSCR fields selected by mask

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XFL

mtfsfi_x

mtfsfi_x

Move to FPSCR Field Immediate (x'FC00 010C')

mtfsfi

mtfsfi.

crfD,IMM

crfD,IMM

(Rc = '0')

(Rc = '1')



FPSCR[crfD] ← IMM

The value of the IMM field is placed into FPSCR field **crfD**.

FPSCR[FX] is altered only if **crfD** = '0'.

When FPSCR[0:3] is specified, bit 0 (FX) and bit 3 (OX) are set to the values of IMM[0] and IMM[3] (that is, even if this instruction causes OX to change from '0' to '1', FX is set from IMM[0] and not by the usual rule that FX is set when an exception bit changes from '0' to '1'). Bit 1 (FEX) and bit 2 (VX) are set according to the usual rule and not from IMM[1:2].

- Other registers altered:
- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
 - Floating-Point Status and Control Register
Affected: FPSCR field **crfD**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

mtmsr

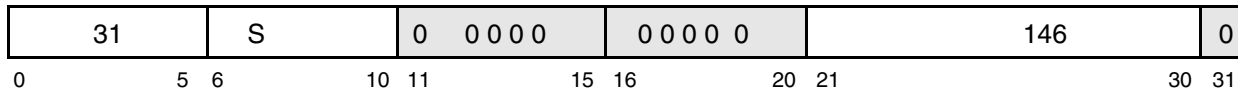
Move to Machine State Register (x'7C00 0124')

mtmsr

mtmsr

rS

 Reserved



MSR ← (rS)

The contents of rS are placed into the MSR.

This is a supervisor-level instruction. It is also an execution synchronizing instruction except with respect to alterations to the POW and LE bits. See the synchronization requirements for special registers and for lookaside buffers section in the the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

In addition, alterations to the MSR[EE] and MSR[RI] bits are effective as soon as the instruction completes. Thus if MSR[EE] = '0' and an external or decremter exception is pending, executing an **mtmsr** instruction that sets MSR[EE] = '1' causes the external or decremter exception to be taken before the next instruction is executed, if no higher priority exception exists.

Other registers altered:

- MSR

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

mtspr

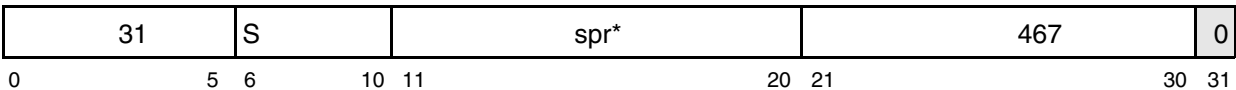
Move to Special-Purpose Register (x'7C00 03A6')

mtspr

mtspr

SPR,rS

 Reserved



***Note:** This is a split field.

$$n \leftarrow \text{spr}[5-9] \parallel \text{spr}[0-4]$$

$$\text{SPR}(n) \leftarrow (\text{rS})$$

The SPR field denotes a special-purpose register. See *Table 2-36. SPR Encodings for PowerPC-Defined Registers* on page 99 and *Table 2-37. SPR Encodings for Espresso-Defined Registers* on page 101 for the spr encodings. The contents of rS are placed into the designated special-purpose register.

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

The value of SPR[0] equals '1' only if writing the register is a supervisor-level operation. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = '1' results in a privileged instruction type program exception.

If MSR[PR] = '1', the only effect of executing an instruction with an SPR number that is not shown in *Table 2-36* on page 99 or *Table 2-37* on page 101 and has SPR[0] = '1' is to cause a privileged instruction type program exception or an illegal instruction type program exception. For all other cases, MSR[PR] = '0' or SPR[0] = '0'. If the SPR field contains any value that is not shown in *Table 2-36* or *Table 2-37*, either an illegal instruction type program exception occurs or the results are boundedly undefined.

Other registers altered:

- See *Table 2-36* on page 99 or *Table 2-37* on page 101.

Simplified mnemonics:

mtxer	rD	equivalent to	mtspr	1,rD
mtlr	rD	equivalent to	mtspr	8,rD
mtctr	rD	equivalent to	mtspr	9,rD

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			XFX

mtsr

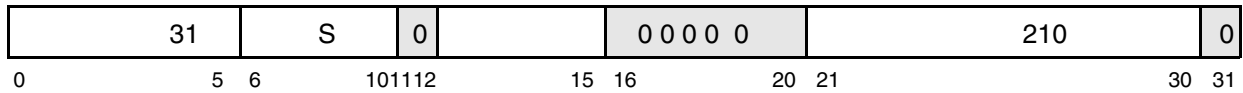
mtsr

Move to Segment Register (x'7C00 01A4')

mtsr

SR,rS

 Reserved



SEGREG(SR) ← (rS)

The contents of rS are placed into SR.

This is a supervisor-level instruction.

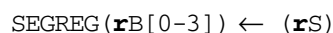
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

mtsrin

rS,rB



- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	Yes			X

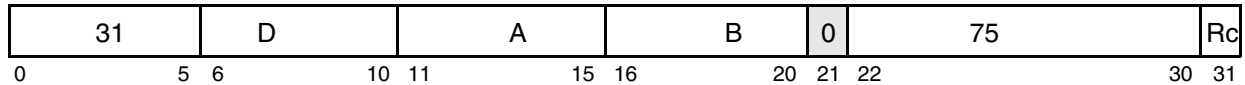
mulhw_x

Multiply High Word (x'7C00 0096')

mulhw_x

mulhw rD,rA,rB (Rc = '0')

mulhw. rD,rA,rB (Rc = '1')

 Reserved


```
prod[0-63] ← (rA) × (rB)
rD ← prod[0-31]
```

The 64-bit product is formed from the contents **rA** and **rB**. The high-order 32 bits of the 64-bit product of the operands are placed into **rD**. Both the operands and the product are interpreted as signed integers.

This instruction might execute faster on some implementations if **rB** contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

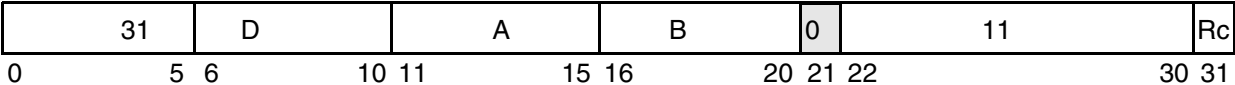
mulhwu_x

Multiply High Word Unsigned (x'7C00 0016')

mulhwu_x

mulhwu **rD,rA,rB** (Rc = '0')
mulhwu. **rD,rA,rB** (Rc = '1')

 Reserved



```
prod[0-63] ← (rA) × (rB)
rD ← prod[0-31]
```

The 32-bit operands are the contents **rA** and **rB**. The high-order 32 bits of the 64-bit product of the operands are placed into **rD**.

Both the operands and the product are interpreted as unsigned integers, except that if Rc = '1', the first three bits of CR0 field are set by signed comparison of the result to zero.

This instruction might execute faster on some implementations if **rB** contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

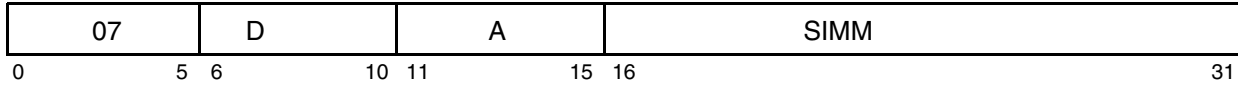
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

mulli

Multiply Low Immediate (x'1C00 0000')

mulli

mulli **rD,rA,SIMM**



```

prod[0-63] ← (rA) × EXTS(SIMM)
rD ← prod[32-63]

```

The first operand is (**rA**). The second operand is the sign-extended value of the SIMM field. The low-order 32-bits of the 64-bit product of the operands are placed into **rD**.

Both the operands and the product are interpreted as signed integers. The low-order 32-bits of the product are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with **mulhdx** or **mulhwx** to calculate a full 64-bit product.

Other registers altered:

- None

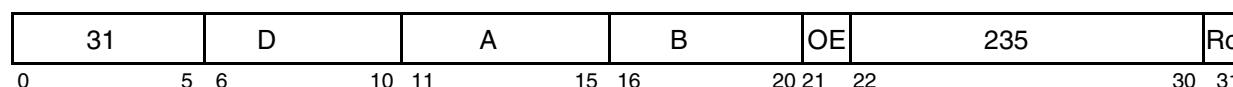
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

mullw_x

Multiply Low Word (x'7C00 01D6')

mullw_x

mullw	rD,rA,rB	(OE = '0' Rc = '0')
mullw.	rD,rA,rB	(OE = '0' Rc = '1')
mullwo	rD,rA,rB	(OE = '1' Rc = '0')
mullwo.	rD,rA,rB	(OE = '1' Rc = '1')



$$\text{prod}[0-63] \leftarrow (\mathbf{rA}) \times (\mathbf{rB})$$

$$\mathbf{rD} \leftarrow \text{prod}[32-63]$$

The 32-bit operands are the contents of **rA** and **rB**. The low-order 32-bits of the 64-bit product $(\mathbf{rA}) \times (\mathbf{rB})$ are placed into **rD**.

The low-order 32-bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

If OE = '1', OV is set if the product cannot be represented in 32 bits. Both the operands and the product are interpreted as signed integers.

This instruction can be used with **mulhwx** to calculate a full 64-bit product.

Note: This instruction might execute faster on some implementations if **rB** contains the operand having the smaller absolute value.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: SO, OV (if OE = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO



nand_x

nand_x

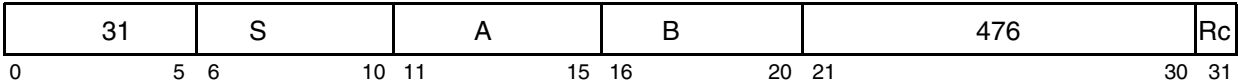
NAND (x'7C00 03B8')

nand

nand.

rA,rS,rB
rA,rS,rB

(Rc = '0')
(Rc = '1')



$rA \leftarrow \neg ((rS) \& (rB))$

The contents of **rS** are ANDed with the contents of **rB** and the complemented result is placed into **rA**.

nand with **rS = rB** can be used to obtain the ones complement.

Other registers altered:

- Condition Register (CR0 field)

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

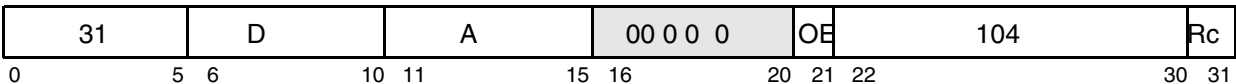
neg_x

Negate (x'7C00 00D0')

neg_x

neg	rD,rA	(OE = '0' Rc = '0')
neg.	rD,rA	(OE = '0' Rc = '1')
nego	rD,rA	(OE = '1' Rc = '0')
nego.	rD,rA	(OE = '1' Rc = '1')

 Reserved



$$rD \leftarrow \neg (rA) + 1$$

The value 1 is added to the complement of the value in **rA**, and the resulting twos complement is placed into **rD**.

If **rA** contains the most negative 32-bit number (x'8000_0000'), the result is the most negative number.
If OE = '1', OV is set.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')
- XER
Affected: SO OV (if OE = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO



nor_x

nor_x

NOR (x'7C00 00F8')

nor

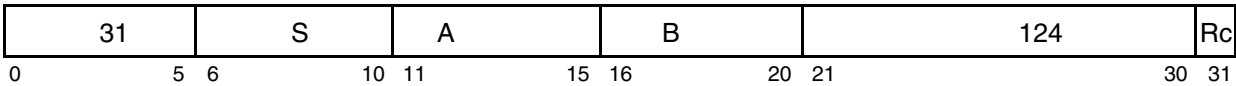
nor.

rA,rS,rB

rA,rS,rB

(Rc = '0')

(Rc = '1')



$$rA \leftarrow \neg ((rS) \mid (rB))$$

The contents of **rS** are ORed with the contents of **rB** and the complemented result is placed into **rA**.

nor with **rS** = **rB** can be used to obtain the ones complement.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

not

rD,rS

equivalent to

nor

rA,rS,rS

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

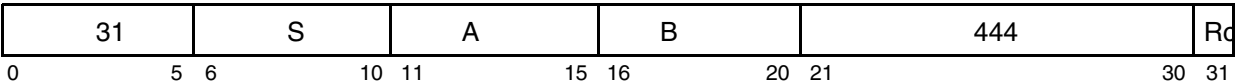
or_x

OR (x'7C00 0378')

or_x

or rA,rS,rB (Rc = '0')

or. rA,rS,rB (Rc = '1')



$$rA \leftarrow (rS) \mid (rB)$$

The contents of **rS** are ORed with the contents of **rB**, and the result is placed into **rA**.

The example under simplified mnemonic **mr** demonstrates the use of the **or** instruction to move register contents.

- Other registers altered:
- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

mr rA,rS equivalent to **or** rA,rS,rS

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X



orc_x

orc_x

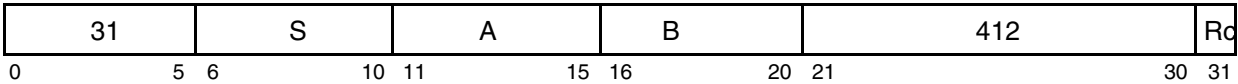
OR with Complement (x'7C00 0338')

orc

orc.

rA,rS,rB
rA,rS,rB

(Rc = '0')
(Rc = '1')



$rA \leftarrow (rS) \mid \neg (rB)$

The contents of rS are ORed with the complement of the contents of rB, and the result is placed into rA.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

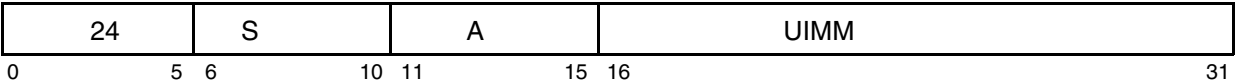
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

ori

ori

OR Immediate (x'6000 0000')

ori **rA,rS,UIMM**



$$rA \leftarrow (rS) \mid ((16)0 \mid \mid UIMM)$$

The contents of **rS** are ORed with 0x0000 || UIMM, and the result is placed into **rA**.

The preferred no-op (an instruction that does nothing) is **ori 0,0,0**.

Other registers altered:

- None

Simplified mnemonics:

nop equivalent to **ori** **0,0,0**

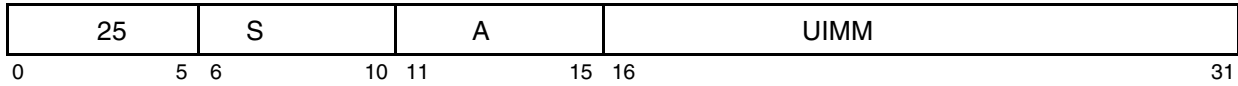
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

oris

oris

OR Immediate Shifted (x'6400 0000')

oris **rA,rS,UIMM**



$$rA \leftarrow (rS) \mid (UIMM \mid (16)0)$$

The contents of **rS** are ORed with UIMM || 0x0000, and the result is placed into **rA**.

Other registers altered:

- None

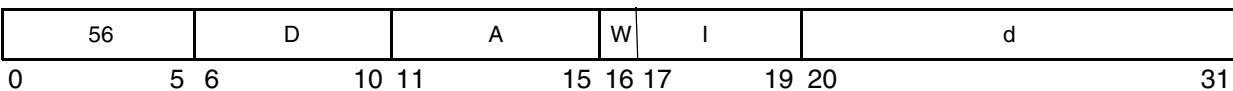
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

psq_l

Paired Single Quantized Load, (x'E000 0000')

psq_l

psq_l **frD,d(rA),W,I**



```

if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
lt ← qri[LD_TYPE]
ls ← qri[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 1
if lt = (5|7) then c ← 2
if W = 0 then
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← dequantized(MEM(EA+c,c),lt,ls)
else
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← 1.0
    
```

PS0 and PS1 in **frD** are loaded with a pair of single-precision floating-point numbers.

Memory is accessed at the effective address (EA is the sum [rA]0 + d) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers, and the results are placed into PS0 and PS1. However, if W = '1', only one number is accessed from memory, converted according to GQR, and placed into PS0. PS1 is loaded with a floating-point value of 1.0.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating-point format. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

Other registers altered:

- None

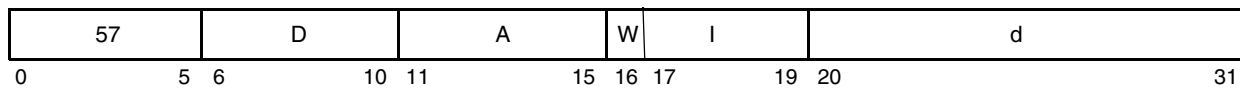
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_lu

Paired Single Quantized Load with Update, (x'E400 0000')

psq_lu

psq_lu frD,d(rA),W,I



```

if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
EA ← (rA) + EXTS(d)
lt ← qri[LD_TYPE]
ls ← qri[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 1
if lt = (5|7) then c ← 2
if W = 0
then
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← dequantized(MEM(EA+c,c),lt,ls)
else
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← 1.0
rA ← EA
    
```

PS0 and PS1 in frD are loaded with a pair of single-precision floating-point numbers.

Memory is accessed at the effective address (EA is the sum [rA] + d) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers, and the results are placed into PS0 and PS1. However, if W = '1', only one number is accessed from memory, converted according to GQR, and placed into PS0. PS1 is loaded with a floating-point value of 1.0.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating-point format. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

The effective address is placed into rA.

If rA = '0', the instruction form is invalid.

Other registers altered:

- None

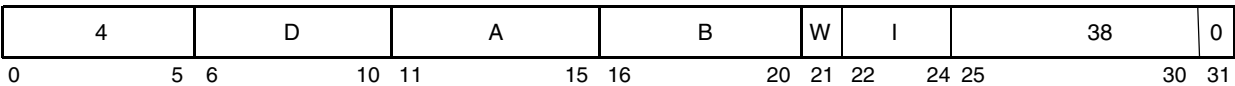
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_lux

psq_lux

Paired Single Quantized Load with update Indexed, (x'1000 004C')

psq_lux frD,rA,rB,W,I



```

if (HID2[PSE] = 0) then Goto illegal instruction error handler
EA ← (rA) + (rB)
lt ← qqrI[LD_TYPE]
ls ← qqrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 1
if lt = (5|7) then c ← 2
if W = 0
then
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← dequantized(MEM(EA+c,c),lt,ls)
else
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← 1.0
rA←EA
    
```

PS0 and PS1 in frD are loaded with a pair of single-precision floating-point numbers.

Memory is accessed at the effective address (EA is the sum [rA] + [rB]) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers and the results are placed into PS0 and PS1. However, if W = '1', only one number is accessed from memory, converted according to GQR, and placed into PS0. PS1 is loaded with a floating-point value of 1.0.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating-point format. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

The effective address is placed into register rA.

If rA = '0', the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_lx

Paired Single Quantized Load Indexed, (x'1000 000C')

psq_lx

psq_lx

frD, rA, rB, W, I

4		D		A		B		W	I	6		0	
0	5	6	10	11	15	16	20	21	22	24	25	30	31

```

if HID2[PSE] = 0 then Goto illegal instruction error handler
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
lt ← qqrI[LD_TYPE]
ls ← qqrI[LD_SCALE]
c ← 4
if lt = (4|6) then c ← 1
if lt = (5|7) then c ← 2
if W = 0 then
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← dequantized(MEM(EA+c,c),lt,ls)
else
    frD_ps0 ← dequantized(MEM(EA,c),lt,ls)
    frD_ps1 ← 1.0

```

PS0 and PS1 in frD are loaded with a pair of single-precision floating-point numbers.

Memory is accessed at the effective address (EA is the sum [rA]0 + [rB]) as defined by the instruction. A pair of numbers from memory are converted as defined by the indicated GQR control registers, and the results are placed into PS0 and PS1. However, if W = '1', only one number is accessed from memory, converted according to GQR, and placed into PS0. PS1 is loaded with a floating-point value of 1.0.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the LOAD_SCALE and the LD_TYPE fields are used. The LD_TYPE field defines whether the data in memory is floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The LOAD_SCALE field is applied only to integer numbers and is a signed integer that is subtracted from the exponent after the integer number from memory has been converted to floating-point format. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

Other registers altered:

- None

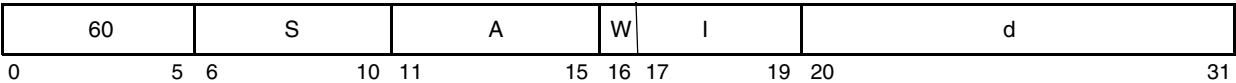
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_st

Paired Single Quantized Store, (x'F000 0000')

psq_st

psq_st $frS, d(rA), W, I$



```

if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
stt ← qqrI[ST_TYPE]
sts ← qqrI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 1
if stt = (5|7) then c ← 2
if W = 0 then
    MEM(EA, c) ← quantized(frS_ps0, stt, sts)
    MEM(EA+c, c) ← quantized(frS_ps1, stt, sts)
else
    MEM(EA, c) ← quantized(frS_ps0, stt, sts)
    
```

The effective address is the sum of (rA|0) + d as defined by the instruction. If W = '1', only one floating-point number from frS_ps0 is quantized and stored to memory starting at the effective address. If W = '0', a pair of floating-point numbers from frS_ps0 and frS_ps1 are quantized and stored to memory starting at the effective address.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating-point number before it is converted to integer and stored to memory. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

For floating-point numbers stored to memory, the addition of the STORE_SCALE field to the exponent does not take place.

Other registers altered:

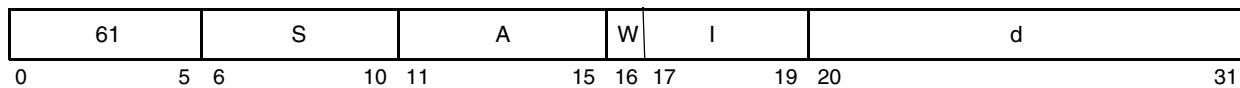
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_stu

Paired Single Quantized Store with update, (x' F400 0000')

psq_stu

psq_stu **frS,d(rA),W,I**


```

if HID2[PSE] = 0 | HID2[LSQE] = 0 then Goto illegal instruction error handler
EA ← (rA) + EXTS(d)
stt ← qriI[ST_TYPE]
sts ← qriI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 1
if stt = (5|7) then c ← 2
if W = 0 then
    MEM(EA,c) ← quantized(frS_ps0,stt,sts)
    MEM(EA+c,c) ← quantized(frS_ps1,stt,sts)
else
    MEM(EA,c) ← quantized(frS_ps0,stt,sts)
rA ← EA
    
```

The effective address is the sum of (rA) + d as defined by the instruction. If W = '1', only one floating-point number from frS_ps0 is quantized and stored to memory starting at the effective address. If W = '0', a pair of floating-point numbers from frS_ps0 and frS_ps1 are quantized and stored to memory starting at the effective address.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating-point number before it is converted to integer and stored to memory.

For floating-point numbers stored to memory, the addition of the STORE_SCALE field to the exponent field does not take place. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

The effective address is placed into register rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		DW

psq_stux

psq_stux

Paired Single Quantized Store with Update Indexed, (x'1000 004E')

psq_stux **frS, rA, rB, W, I**

4		S		A		B		W	I	39		0	
0	5	6	10	11	15	16	20	21	22	24	25	30	31

```

if HID2[PSE] = 0 then Goto illegal instruction error handler
EA ← (rA) + (rB)
stt ← qriI[ST_TYPE]
sts ← qriI[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 1
if stt = (5|7) then c ← 2
if W = 0 then
    MEM(EA, c) ← quantized(frS_ps0, stt, sts)
    MEM(EA+c, c) ← quantized(frS_ps1, stt, sts)
else
    MEM(EA, c) ← quantized(frS_ps0, stt, sts)
rA ← EA

```

The effective address is the sum of (rA) + (rB) as defined by the instruction. If W = '1', only one floating-point number from frS_ps0 is quantized and stored to memory starting at the effective address. If W = '0', a pair of floating-point numbers from frS_ps0 and frS_ps1 are quantized and stored to memory starting at the effective address.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating-point number before it is converted to integer and stored to memory. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

For floating-point numbers stored to memory, the addition of the STORE_SCALE field to the exponent field does not take place.

The effective address is placed into rA. If rA = 0, the instruction form is invalid.

Other registers altered:

- None

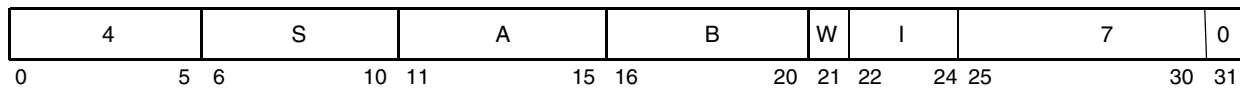
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		XW

psq_stx

Paired Single Quantized Store Indexed, (x'1000 000E')

psq_stx

psq_stx frS,rA,rB,W,I



```

if HID2[PSE] = 0 then Goto illegal instruction error handler
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
stt ← qri[ST_TYPE]
sts ← qri[ST_SCALE]
c ← 4
if stt = (4|6) then c ← 1
if stt = (5|7) then c ← 2
if W = 0 then
    MEM(EA,c) ← quantized(frS_ps0,stt,sts)
    MEM(EA+c,c) ← quantized(frS_ps1,stt,sts)
else
    MEM(EA,c) ← quantized(frS_ps0,stt,sts)

```

The effective address is the sum of (rA|0) + (rB) as defined by the instruction. If W = '1', only one floating-point number from frS_ps0 is quantized and stored to memory starting at the effective address. If W = '0', a pair of floating-point numbers from frS_ps0 and frS_ps1 is quantized and stored to memory starting at the effective address.

The 3-bit field I selects one of the eight 32-bit GQR control registers. From this register, the STORE_SCALE and the ST_TYPE fields are used. The ST_TYPE field defines whether the data stored to memory is to be floating-point or integer format. If the latter, it also defines whether each integer is 8 bits or 16 bits, signed or unsigned. The STORE_SCALE field is a signed integer that is added to the exponent of the floating-point number before it is converted to integer and stored to memory. (See *Paired Single Load and Store Instructions* on page 92 for dequantized operation.)

For floating-point numbers stored to memory, the addition of the STORE_SCALE field to the exponent field does not take place.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		XW

ps_abs_x

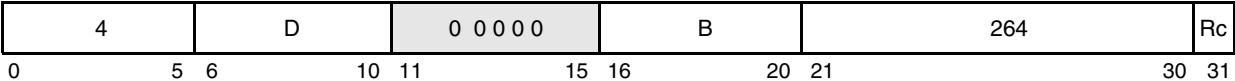
Paired Single Absolute Value (x'1000 0210')

ps_abs_x

ps_absfrD,frB(Rc = 0)

ps_abs.frD,frB(Rc = '1')

Reserved



If HID2[PSE] = 0 then invoke the illegal instruction error handler

frD_ps0 ← b'0' || (frB_ps0)[1-31]

frD_PS1 ← b'0' || (frB_ps1)[1-31]

The contents of frB_ps0 with bit 0 cleared are placed into frD_ps0.

The contents of frB_ps1 with bit 0 cleared are placed into frD_ps1.

Note: The ps_abs instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN can be altered by ps_abs. This instruction does not alter the FPSCR.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

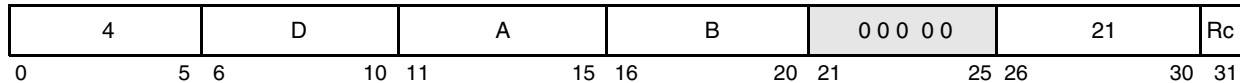
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_addx

Paired Single Add (x'1000 002A')

ps_addx

ps_add **frD,frA,frB** (Rc = 0)
ps_add. **frD,frA,frB** (Rc = '1')

☐ Reserved


If HID2[PSE] = 0 then invoke the illegal instruction error handler
 $\mathbf{frD_ps0} \leftarrow (\mathbf{frA_ps0}) + (\mathbf{frB_ps0})$
 $\mathbf{frD_ps1} \leftarrow (\mathbf{frA_ps1}) + (\mathbf{frB_ps1})$

The floating-point operand in **frA_ps0** is added to the floating-point operand in **frB_ps0**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in **frA_ps1** is added to the floating-point operand in **frB_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by 1 for each bit shifted until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 25 bits in the significand and all 3 guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right 1-bit position, and the exponent is increased by 1. FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
 Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
 Affected: FPRF(ps0 only), FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

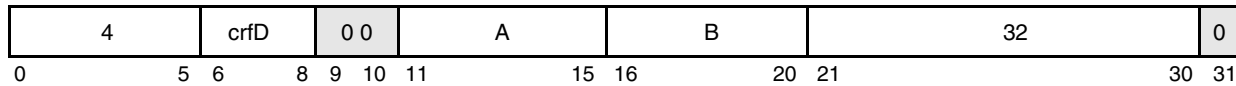
ps_cmpo0

Paired Singles Compare Ordered High (x'1000 0040')

ps_cmpo0

ps_cmpo0

crfD,frA,frB

☐ Reserved

```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
if ((frA_ps0) is a NaN or (frB_ps0)) is a NaN ) then
    c ← 0b0001
else if (frA_ps0) < (frB_ps0) then
    c ← 0b1000
else if (frA_ps0) > (frB_ps0) then
    c ← 0b0100
else
    c ← 0b0010
FPCC ← c
CR[(4 × crfD), (4 × crfD + 3)] ← c

if ((frA_ps0) is an SNaN or (frB_ps0) is an SNaN) then
    VXSNaN ← 1
if VE = 0 then
    VXVC ← 1
else if ((frA_ps0) is a QNaN or (frB_ps0) is a QNaN ) then
    VXVC ← 1

```

The floating-point operand in **frA_ps0** is compared to the floating-point operand in **frB_ps0**. The result of the compare is placed into CR field **crfD** and the FPCC.

If one of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, VXSNaN is set. If invalid operation is disabled (VE = '0'), VXVC is set. Otherwise, if one of the operands is a QNaN, VXVC is set.

Other registers altered (exception conditions are based on ps0 values):

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC (ps0 only), FX, VXSNaN, VXVC

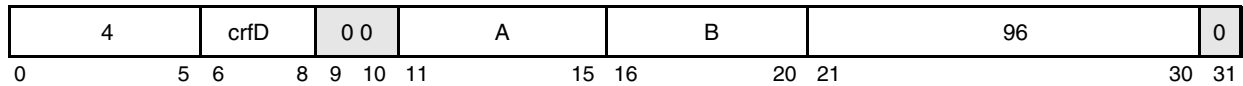
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_cmpo1

Paired Singles Compare Ordered Low (x'1000 00C0')

ps_cmpo1

ps_cmpo1 crfD,frA,frB

☐ Reserved


```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA_ps1) is a NaN or (frB_ps1) is a NaN ) then
    c ← 0b0001
else if ((frA_ps1) < (frB_ps1)) then
    c ← 0b1000
else if ((frA_ps1) > (frB_ps1)) then
    c ← 0b0100
else
    c ← 0b0010
FPCC ← c
CR[(4 × crfD), (4 × crfD + 3)] ← c

if ((frA_ps1) is an SNaN or (frB_ps1) is an SNaN) then
    VXSNaN ← 1
if VE = 0 then
    VXVC ← 1
else if ((frA_ps1)) is a QNaN or (frB_ps1)) is a QNaN) then
    VXVC ← 1

```

The floating-point operand in **frA_ps1** is compared to the floating-point operand in **frB_ps1**. The result of the compare is placed into CR field **crfD** and the FPCC.

If one of the operands is a NaN, either quiet or signaling, CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, VXSNaN is set. If invalid operation is disabled (VE = '0'), VXVC is set. Otherwise, if one of the operands is a QNaN, VXVC is set.

Other registers altered: (exception conditions are based on ps1 values)

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC (ps1 only), FX, VXSNaN, VXVC

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

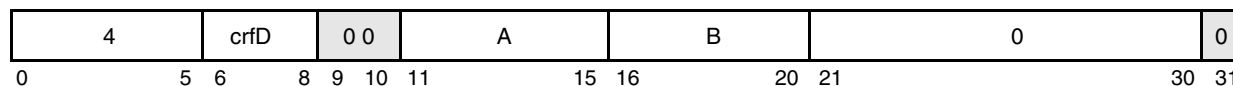
ps_cmpu0

Paired Singles Compare Unordered High (x'1000 0000')

ps_cmpu0

ps_cmpu0

crfD,frA,frB

☐ Reserved

```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
if (( frA_ps0) is a NaN or ( frB_ps0) is a NaN ) then
    c ← 0b0001
else if ( frA_ps0) < ( frB_ps0) then
    c ← 0b1000
else if ( frA_ps0) > ( frB_ps0) then
    c ← 0b0100
else
    c ← 0b0010

FPCC ← c
CR[(4 × crfD), (4 × crfD + 3)] ← c

if (( frA_ps0) is an SNaN or ( frB_ps0) is an SNaN ) then
    VXSNaN ← 1

```

The floating-point operand in **frA_ps0** is compared to the floating-point operand in **frB_ps0**. The result of the compare is placed into CR field **crfD** and the FPCC.

If one of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, VXSNaN is set.

Other registers altered (exception conditions are based on ps0 values):

- Condition Register (CR field specified by operand **crfD**)
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC (ps0 only), FX, VXSNaN

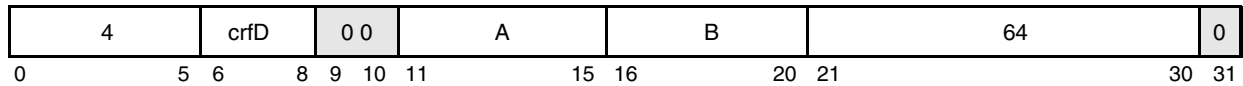
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_cmpu1

Paired Singles Compare Unordered Low(x'1000 0080')

ps_cmpu1

ps_cmpu1 crfD,frA,frB

☐ Reserved


```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
if ((frA_ps1) is a NaN or (frB_ps1) is a NaN) then
    c ← 0b0001
else if (frA_ps1) < (frB_ps1) then
    c ← 0b1000
else if (frA_ps1) > (frB_ps1) then
    c ← 0b0100
else
    c ← 0b0010
FPCC ← c
CR[(4 × crfD), (4 × crfD + 3)] ← c

if ((frA_ps1) is an SNaN or (frB_ps1) is an SNaN ) then
    VXSNaN ← 1
    
```

The floating-point operand in **frA_ps1** is compared to the floating-point operand in **frB_ps1**. The result of the compare is placed into CR field **crfD** and the FPCC.

If one of the operands is a NaN, either quiet or signaling, the CR field **crfD** and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, VXSNaN is set.

Other registers altered (exception conditions are based on ps1 values):

- Condition Register (CR field specified by operand **crfD**):
Affected: LT, GT, EQ, UN
- Floating-Point Status and Control Register
Affected: FPCC (ps1 only), FX, VXSNaN

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

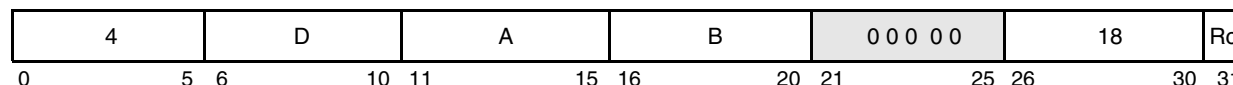
ps_div_x

Paired Single Divide (x'1000 0024')

ps_div_x

ps_div **frD,frA,frB** (Rc = '0')

ps_div. **frD,frA,frB** (Rc = '1')

 Reserved


If HID2[PSE] = 0 then invoke the illegal instruction error handler

frD_ps0 ← ((**frA_ps0**) ÷ (**frB_ps0**))

frD_ps1 ← ((**frA_ps1**) ÷ (**frB_ps1**))

The floating-point operand in register **frA_ps0** is divided by the floating-point operand in register **frB_ps0**. The remainder is not supplied as a result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is divided by the floating-point operand in register **frB_ps1**. The remainder is not supplied as a result. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to single precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, ZX, XX, VXSNaN, VXIDI, VXZDZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

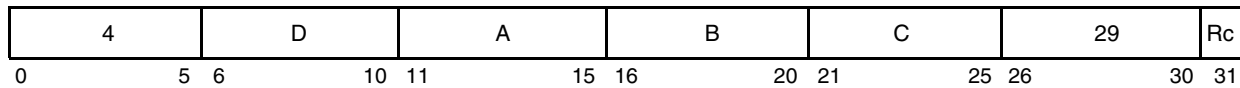
ps_madd_x

Paired Single Multiply-Add (x'1000 003A')

ps_madd_x

ps_madd **frD,frA,frC,frB** (**Rc** = '0')

ps_madd. **frD,frA,frC,frB** (**Rc** = '1')



If **HID2[PSE]** = 0 then invoke the illegal instruction error handler

$$\mathbf{frD_ps0} \leftarrow ((\mathbf{frA_ps0}) \times (\mathbf{frC_ps0})) + (\mathbf{frB_ps0})$$

$$\mathbf{frD_ps1} \leftarrow ((\mathbf{frA_ps1}) \times (\mathbf{frC_ps1})) + (\mathbf{frB_ps1})$$

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps0** is added to this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field **RN** of the **FPSCR** and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps1** is added to this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field **RN** of the **FPSCR** and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the **ps0** result, except for invalid operation exceptions when **FPSCR[VE]** = '1'.

Other registers altered (exception conditions are based on either **ps0** or **ps1** values):

- Condition Register (CR1 field)
Affected: **FX**, **FEX**, **VX**, **OX** (if **Rc** = '1')
- Floating-Point Status and Control Register (**FPSCR**)
Affected: **FPRF** (**ps0** only), **FR**, **FI**, **FX**, **OX**, **UX**, **XX**, **VXSNAN**, **VXISI**, **VXIMZ**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

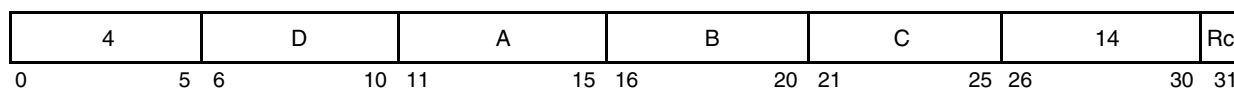
ps_madds0x

Paired Single Multiply-Add Scalar high(x'1000 001C')

ps_madds0x

ps_madds0 **frD,frA,frC,frB** (Rc = '0')

ps_madds0. **frD,frA,frC,frB** (Rc = '1')



```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← ((frA_ps0) × (frC_ps0)) + (frB_ps0)
frD_ps1 ← ((frA_ps1) × (frC_ps0)) + (frB_ps1)

```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps0** is added to this intermediate result. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and is placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps1** is added to this intermediate result. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and is placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

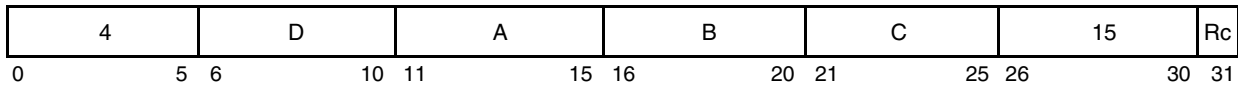
ps_madds1x

Paired Single Multiply-Add Scalar low(x'1000 001E')

ps_madds1x

ps_madds1 frD,frA,frC,frB (Rc = '0')

ps_madds1. frD,frA,frC,frB (Rc = '1')



```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frA_ps0) × (frC_ps1) + (frB_ps0)
frD_ps1 ← (frA_ps1) × (frC_ps1) + (frB_ps1)
```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps0** is added to this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps1** is added to this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

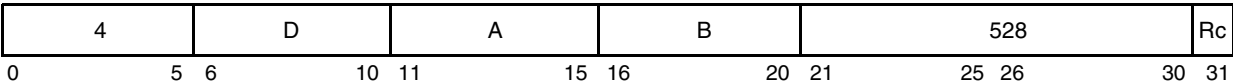
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_merge00_x

Paired Single MERGE high (x'1000 0420')

ps_merge00 **frD,frA,frB** (Rc = '0')
ps_merge00. **frD,frA,frB** (Rc = '1')

ps_merge00_x



```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frA_ps0)
frD_ps1 ← (frB_ps0)
```

The floating-point operand in register **frA**_ps0 is moved to register **frD**_ps0 and floating-point operand in register **frB**_ps0 is moved to register **frD**_ps1.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

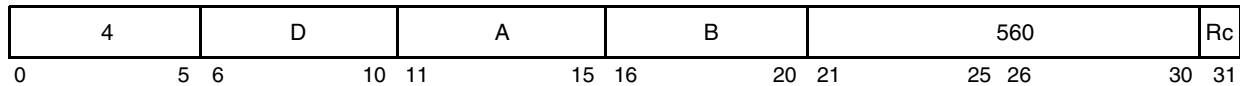
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge01_x

Paired Single MERGE direct(x'1000 0460')

ps_merge01 **frD,frA,frB** (Rc = '0')
ps_merge01. **frD,frA,frB** (Rc = '1')

ps_merge01_x



```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frA_ps0)
frD_ps1 ← (frB_ps1)

```

The floating-point operand in register **frA**_ps0 is moved to register **frD**_ps0 and floating-point operand in register **frB**_ps1 is moved to register **frD**_ps1.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

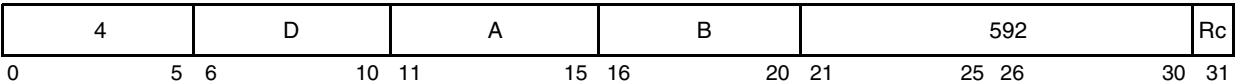
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge10_x

Paired Single MERGE swapped(x'1000 04A0')

ps_merge10_x

ps_merge10 **frD,frA,frB** (Rc = '0')
ps_merge10. **frD,frA,frB** (Rc = '1')



```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frA_ps1)
frD_ps1 ← (frB_ps0)
```

The floating-point operand in register **frA**_ps1 is moved to register **frD**_ps0 and floating-point operand in register **frB**_ps0 is moved to register **frD**_ps1.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

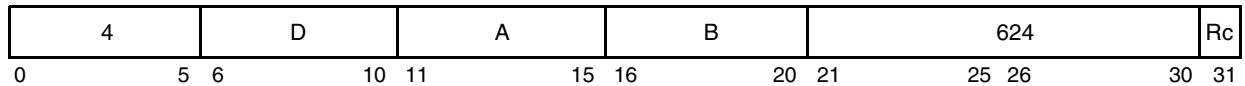
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_merge11_x

Paired Single MERGE low(x'1000 04E0')

ps_merge11 **frD,frA,frB** (Rc = '0')
ps_merge11. **frD,frA,frB** (Rc = '1')

ps_merge11_x



```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frA_ps1)
frD_ps1 ← (frB_ps1)

```

The floating-point operand in register **frA**_ps1 is moved to register **frD**_ps0 and floating-point operand in register **frB**_ps1 is moved to register **frD**_ps1.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

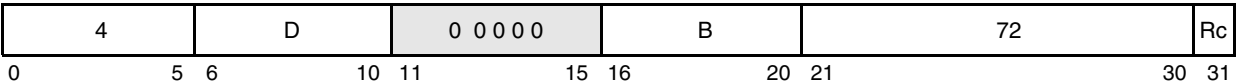
ps_mr_x

Paired Single Move Register (x'1000 0090')

ps_mr_x

ps_mr frD,frB (Rc = '0')
ps_mr. frD,frB (Rc = '1')

 Reserved



If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (frB_ps0)
frD_ps1 ← (frB_ps1)

The contents of register frB_ps0 are placed into frD_ps0.
The contents of register frB_ps1 are placed into frD_ps1.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_msub_x

Paired Single Multiply-Subtract (x'1000 0038')

ps_msub_x

ps_msub **frD,frA,frC,frB** (**Rc** = '0')

ps_msub. **frD,frA,frC,frB** (**Rc** = '1')

4	D	A	B	C	28	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← [(frA_ps0) × (frC_ps0)] - (frB_ps0)
frD_ps1 ← [(frA_ps1) × (frC_ps1)] - (frB_ps1)

```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps0** is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps1** is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

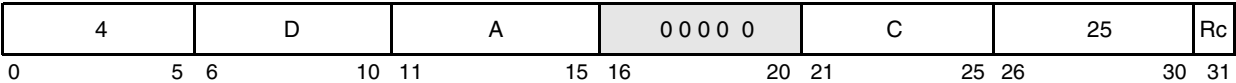
ps_mulx

Paired Single Multiply (x'1000 0032')

ps_mulx

ps_mul frD,frA,frC (Rc = '0')
ps_mul. frD,frA,frC (Rc = '1')

Reserved



```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
    frD_ps0 ← ((frA_ps0) × (frC_ps0))
    frD_ps1 ← ((frA_ps1) × (frC_ps1))
```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

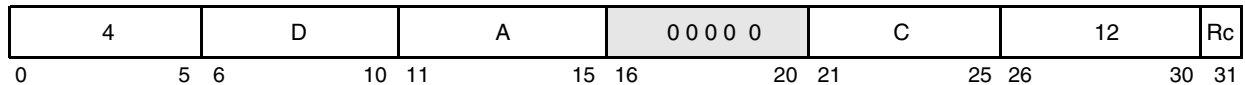
ps_muls0_x

Paired Single Multiply Scalar High (x'1000 0018')

ps_muls0_x

ps_muls0 frD,frA,frC (Rc = '0')

ps_muls0. frD,frA,frC (Rc = '1')

☐ Reserved


```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← ((frA_ps0) × (frC_ps0))
frD_ps1 ← ((frA_ps1) × (frC_ps0))
```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps0**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

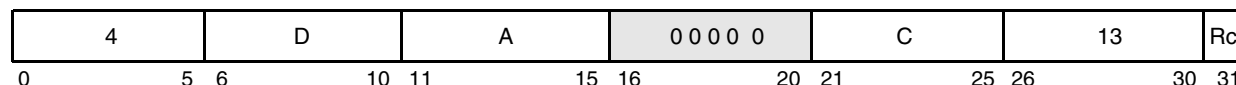
ps_muls1_x

Paired Single Multiply Scalar Low (x'1000 001A')

ps_muls1_x

ps_muls1 **frD,frA,frC** (Rc = '0')

ps_muls1. **frD,frA,frC** (Rc = '1')

☐ Reserved


if HID2[PSE] = 0 then invoke the illegal instruction error handler

frD_ps0 ← ((**frA_ps0**) × (**frC_ps1**))

frD_ps1 ← ((**frA_ps1**) × (**frC_ps1**))

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

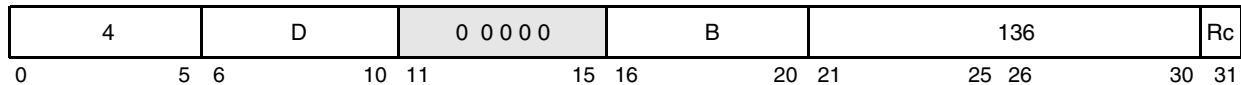
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_nabs_x

Paired Single Negative Absolute Value (x'1000 0110')

ps_nabs **frD,frB** (Rc = '0')
ps_nabs. **frD,frB** (Rc = '1')

ps_nabs_x

☐ Reserved


If HID2[PSE] = 0 then invoke the illegal instruction error handler

frD_ps0 ← b'1' || (**frB_ps0**) [1-31]

frD_ps1 ← b'1' || (**frB_ps1**) [1-31]

The contents of register **frB_ps0** with bit 0 set are placed into **frD_ps0**.

The contents of register **frB_ps1** with bit 0 set are placed into **frD_ps1**.

Note: The **ps_nabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN can be altered by **ps_nabs**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

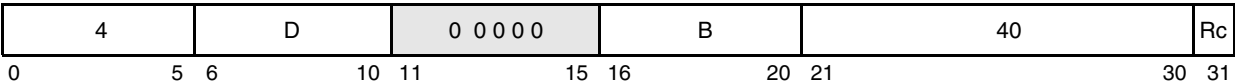
ps_neg_x

Paired Single Negate (x'1000 0050')

ps_neg_x

ps_neg **frD,frB** (Rc = '0')
ps_neg. **frD,frB** (Rc = '1')

 Reserved



If HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← (¬(**frB_ps0**)[0] || (**frB_ps0**)[1-31])
frD_ps1 ← (¬(**frB_ps1**)[0] || (**frB_ps1**)[1-31])

The contents of register **frB_ps0** with bit 0 inverted are placed into **frD_ps0**.
The contents of register **frB_ps1** with bit 0 inverted are placed into **frD_ps1**.

Note: The **ps_neg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **ps_neg**. This instruction does not alter the FPSCR.

Other registers altered:

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		X

ps_nmaddx

Paired Single Negative Multiply-Add (x'1000 003E')

ps_nmaddx

ps_nmadd **frD,frA,frC,frB** (**Rc** = '0')
ps_nmadd. **frD,frA,frC,frB** (**Rc** = '1')

4		D		A		B		C		31		Rc
0	5	6	10	11	15	16	20	21	25	26	30	31

```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← -((frA_ps0) × (frC_ps0) + (frB_ps0) )
frD_ps1 ← -((frA_ps1) × (frC_ps1) + (frB_ps1) )

```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps0** is added to this intermediate product and the result is negated.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps1** is added to this intermediate product and the result is negated.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

This instruction produces the same result as would be obtained by using the Paired Single Multiply-Add (**ps_maddx**) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI, VXIMZ

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

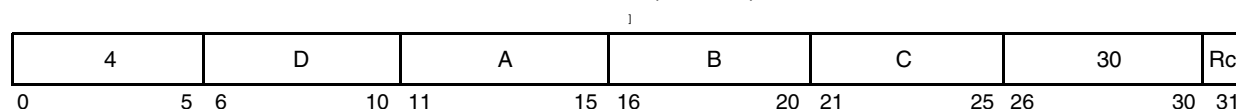
ps_nmsub_x

Paired Single Negative Multiply-Subtract (x'1000 003C')

ps_nmsub_x

ps_nmsub **frD,frA,frC,frB** (Rc = '0')

ps_nmsub. **frD,frA,frC,frB** (Rc = '1')



```

if HID2[PSE] = 0 then invoke the illegal instruction error handler
    frD_ps0 ← -((frA_ps0) × (frC_ps0) - (frB_ps0) )
    frD_ps1 ← -((frA_ps1) × (frC_ps1) - (frB_ps1) )

```

The floating-point operand in register **frA_ps0** is multiplied by the floating-point operand in register **frC_ps0**. The floating-point operand in register **frB_ps0** is subtracted from this intermediate product.

If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, and then negated and placed into **frD_ps0**.

The floating-point operand in register **frA_ps1** is multiplied by the floating-point operand in register **frC_ps1**. The floating-point operand in register **frB_ps1** is subtracted from this intermediate product. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, and then negated and placed into **frD_ps1**.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**ps_msub_x**) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of '0'.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

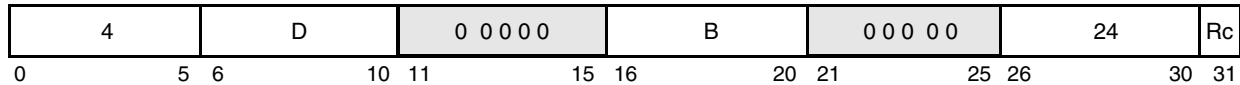
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_resx

Paired Single Reciprocal Estimate (x'1000 0030')

ps_resx

ps_res **frD,frB** (Rc = '0')
ps_res. **frD,frB** (Rc = '1')

☐ Reserved


A single-precision estimate of the reciprocal of the floating-point operand in register **frB_ps0** is placed into register **frD_ps0** and a single-precision estimate of the reciprocal of the floating-point operand in register **frB_ps1** is placed into register **frD_ps1**. These estimates, placed into register **frD_ps0** and **frD_ps1**, are correct to a precision of one part in 4096 of the reciprocal of **frB_ps0** and **frB_ps1**, respectively. That is, for each calculation:

$$ABS \left(\frac{estimate - \left(\frac{1}{x} \right)}{\left(\frac{1}{x} \right)} \right) \leq \frac{1}{4096}$$

where x is the **frB_ps0** or **frB_ps1** value in the source registers.

Operation with various special values of the operand is summarized as follows:

Operand	Result	Exception
$-\infty$	-0	None
-0	$-\infty^1$	ZX
$+0$	$+\infty^1$	ZX
$+\infty$	$+0$	None
SNaN	QNaN ²	VXSNAN
QNaN	QNaN	None

1. No result if FPSCR[ZE] = '1'
2. No result if FPSCR[VE] = '1'

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)

Affected: FPRF (ps0 only), FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNaN

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_rsqrte_x

Paired Single Reciprocal Square Root Estimate (x'1000 0034')

ps_rsqrte_x

ps_rsqrte **frD,frB** (Rc = '0')
ps_rsqrte. **frD,frB** (Rc = '1')

☐ Reserved

4	D	0 0 0 0 0	B	0 0 0 0 0	26	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

A single-precision estimate of the reciprocal of the square root of the floating-point operand in register **frB_ps0** is placed into register **frD_ps0**. A single-precision estimate of the reciprocal of the square root of the floating-point operand in register **frB_ps1** is placed into register **frD_ps1**. These estimates, placed into register **frD_ps0** and **frD_ps1**, are correct to a precision of one part in 4096 of the reciprocal of the square root of **frB_ps0** and **frB_ps1**. That is, for each calculation:

$$ABS \left(\frac{estimate - \left(\frac{1}{\sqrt{x}} \right)}{\left(\frac{1}{\sqrt{x}} \right)} \right) \leq \frac{1}{4096}$$

where x is the **frB_ps0** or **frB_ps1** value in the source registers.

Operations with various special values of the operand is summarized as follows:

Operand	Result	Exception
$-\infty$	QNaN ²	VXSQRT
<0	QNaN ²	VXSQRT
-0	$-\infty$ ¹	ZX
+0	$+\infty$ ¹	
∞ ¹	ZX	
$+\infty$	+0	None
SNaN	QNaN ²	VXSNAN
QNaN	QNaN	None

1. No result if FPSCR[ZE] = '1'.
2. No result if FPSCR[VE] = '1'.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1' and zero divide exceptions when FPSCR[ZE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR (undefined), FI (undefined), FX, ZX, VXSNaN, VXSQRT

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_sel_x

Paired Single Select (x'1000 002E')

ps_sel_x

ps_sel **frD,frA,frC,frB** (Rc = '0')
ps_sel. **frD,frA,frC,frB** (Rc = '1')

4		D		A		B		C		23		Rc
0	5	6	10	11	15	16	20	21	25	26	30	31

```

If HID2[PSE] = 0 then invoke the illegal instruction error handler
if (frA_ps0) ≥ 0.0 then
    frD_ps0 ← (frC_ps0)
else
    frD_ps0 ← (frB_ps0)
if (frA_ps1) ≥ 0.0 then
    frD_ps1 ← (frC_ps1)
else
    frD_ps1 ← (frB_ps1)

```

The floating-point operand in register **frA**_ps0 is compared to the value '0'. If the operand is greater than or equal to '0', register **frD**_ps0 is set to the contents of register **frC**_ps0. If the operand is less than '0' or is a NaN, register **frD**_ps0 is set to the contents of register **frB**_ps0.

The floating-point operand in register **frA**_ps1 is compared to the value '0'. If the operand is greater than or equal to '0', register **frD**_ps1 is set to the contents of register **frC**_ps1. If the operand is less than '0' or is a NaN, register **frD**_ps1 is set to the contents of register **frB**_ps1.

These comparisons ignore the sign of '0' (that is, regard +0 as equal to −0).

Care must be taken in using **ps_sel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

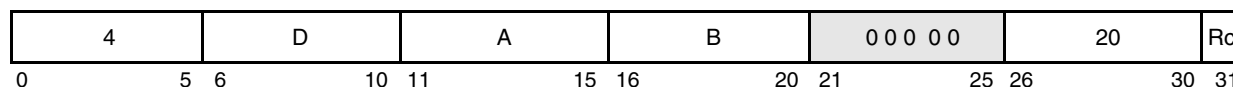
ps_sub_x

Paired Single Subtract (x'1000 0028')

ps_sub_x

ps_sub **frD,frA,frB** (Rc = '0')

ps_sub. **frD,frA,frB** (Rc = '1')

 Reserved


If HID2[PSE] = 0 then invoke the illegal instruction error handler

frD_ps0 ← ((**frA_ps0**) - (**frB_ps0**))

frD_ps1 ← ((**frA_ps1**) - (**frB_ps1**))

The floating-point operand in register **frB_ps0** is subtracted from the floating-point operand in register **frA_ps0**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frB_ps1** is subtracted from the floating-point operand in register **frA_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register (FPSCR)
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

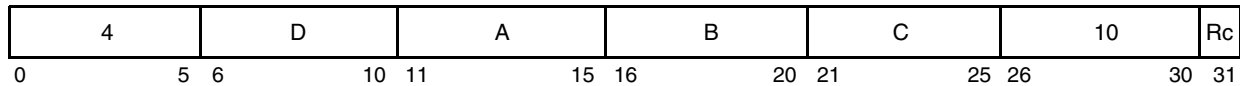
ps_sum0_x

Paired Single vector SUM high (x'1000 0014')

ps_sum0_x

ps_sum0 frD,frA,frC,frB (Rc = '0')

ps_sum0. frD,frA,frC,frB (Rc = '1')



```
if HID2[PSE] = 0 then invoke the illegal instruction error handler
frD_ps0 ← ((frA_ps0) + (frB_ps1))
frD_ps1 ← (frC_ps1)
```

The floating-point operand in register **frA_ps0** is added to the floating-point operand from register **frB_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps0**.

The floating-point operand in register **frC_ps1** is placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps0 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (ps0 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

ps_sum1_x

Paired Single vector SUM low(x'1000 0016')

ps_sum1_x

ps_sum1 **frD,frA,frC,frB** (Rc = '0')

ps_sum1. **frD,frA,frC,frB** (Rc = '1')

4	D	A	B	C	11	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

```

if HID2[PSE] = 0 then Goto illegal instruction error handler
frD_ps0 ← (frC_ps0)
frD_ps1 ← ((frA_ps0) + (frB_ps1))

```

The floating-point operand in register **frC_ps0** is placed into **frD_ps0**.

The floating-point operand in register **frA_ps0** is added to the floating-point operand from register **frB_ps1**. If the most-significant bit of the resultant significand is not a '1', the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **frD_ps1**.

FPSCR[FPRF] is set to the class and sign of the ps1 result, except for invalid operation exceptions when FPSCR[VE] = '1'.

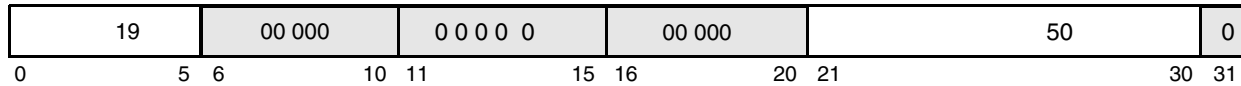
Other registers altered (exception conditions are based on either ps0 or ps1 values):

- Condition Register (CR1 field)
Affected: FX, FEX, VX, OX (if Rc = '1')
- Floating-Point Status and Control Register
Affected: FPRF (ps1 only), FR, FI, FX, OX, UX, XX, VXSNaN, VXISI

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA		Yes		A

rfi
rfi

Return from Interrupt (x'4C00 0064')

☐ Reserved

 $MSR[0, 5-9, 16-23, 25-27, 30-31] \leftarrow SRR1[0, 5-9, 16-23, 25-27, 30-31]$
 $MSR[13] \leftarrow 0$
 $NIA \leftarrow iea\ SRR0[0-29] \parallel 0b00$

Bits SRR1[0, 5:9, 16:23, 25:27, 30:31] are placed into the corresponding bits of the MSR. MSR[13] is set to '0'. If the new MSR value does not enable any pending exceptions, the next instruction is fetched, under control of the new MSR value, from the address SRR0[0:29] || 0b00. If the new MSR value enables one or more pending exceptions, the exception associated with the highest priority pending exception is generated. In this case, the value placed into SRR0 by the exception processing mechanism is the address of the instruction that would have been executed next had the exception not occurred.

This is a supervisor-level, context synchronizing instruction.

Other registers altered:

- MSR

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	YES			XL

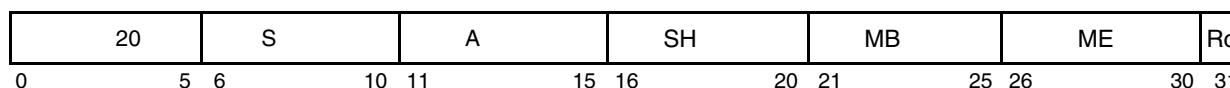
rlwimix

rlwimix

Rotate Left Word Immediate then Mask Insert (x'5000 0000')

rlwimi **rA,rS,SH,MB,ME** (Rc = '0')

rlwimi. **rA,rS,SH,MB,ME** (Rc = '1')



```

n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← (r & m) | (rA & ~m)

```

The contents of **rS** are rotated left the number of bits specified by operand **SH**. A mask is generated having '1' bits from bit **MB** through bit **ME**, and '0' bits elsewhere. The rotated data is inserted into **rA** under control of the generated mask.

The **rlwimi** instruction can be used to copy a bit field of any length from register **rS** into the contents of **rA**. This field can start from any bit position in **rS** and be placed into any position in **rA**. The length of the field can range from 0 - 32 bits. The remaining bits in register **rA** remain unchanged.

- To copy byte_0 (bits 0:7) from **rS** into byte_3 (bits 24:31) of **rA**, set:
SH = 8
MB = 24
ME = 31.
- In general, to copy an *n*-bit field that starts in bit position *b* in register **rS** into register **rA** starting a bit position *c*, set:
SH = 32 - *c* + *b* Mod(32)
MB = *c*
ME = (*c* + *n*) - 1 Mod(32)

Other registers altered:

- Condition Register (CR0 field)
 Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

inslwi rA,rS,n,b equivalent to **rlwimi rA,rS,32 - b,b,b + n - 1**
insrwi rA,rS,n,b (n > 0) equivalent to **rlwimi rA,rS,32 - (b + n),b,(b + n) - 1**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				M

rlwinmx

rlwinmx

Rotate Left Word Immediate then AND with Mask (x'5400 0000')

rlwinm **rA,rS,SH,MB,ME** (**Rc** = '0')

rlwinm. **rA,rS,SH,MB,ME** (**Rc** = '1')

21	S	A	SH	MB	ME	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

```

n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m

```

The contents of **rS** are rotated left the number of bits specified by operand **SH**. A mask is generated having '1' bits from bit **MB** through bit **ME** and '0' bits elsewhere. The rotated data is ANDed with the generated mask, and the result is placed into **rA**.

The **rlwinm** instruction can be used to extract, rotate, shift, and clear bit fields using the following methods:

- To extract an n -bit field, that starts at bit position b in **rS**, right-justified into **rA** (clearing the remaining $32 - n$ bits of **rA**), set:
 $SH = b + n$
 $MB = 32 - n$
 $ME = 31$
- To extract an n -bit field, that starts at bit position b in **rS**, left-justified into **rA** (clearing the remaining $32 - n$ bits of **rA**), set:
 $SH = b$
 $MB = '0'$
 $ME = n - 1$
- To rotate the contents of a register left (or right) by n bits, set:
 $SH = n (32 - n)$
 $MB = '0'$
 $ME = 31$
- To shift the contents of a register right by n bits, set:
 $SH = 32 - n$
 $MB = n$
 $ME = 31$

It can also be used to clear the high-order b bits of a register, and then shift the result left by n bits, set:

```

SH = n
MB = b - n,
ME = 31 - n

```

- To clear the low-order n bits of a register set:
 $SH = '0'$
 $MB = '0'$
 $ME = 31 - n.$

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Simplified mnemonics:

extlwi rA,rS,n,b ($n > 0$)	equivalent to	rlwinm rA,rS,b,0,n – 1
extrwi rA,rS,n,b ($n > 0$)	equivalent to	rlwinm rA,rS,b + n,32 – n,31
rotlwi rA,rS,n	equivalent to	rlwinm rA,rS,n,0,31
rotrwi rA,rS,n	equivalent to	rlwinm rA,rS,32 – n,0,31
slwi rA,rS,n ($n < 32$)	equivalent to	rlwinm rA,rS,n,0,31–n
srwi rA,rS,n ($n < 32$)	equivalent to	rlwinm rA,rS,32 – n,n,31
clrlwi rA,rS,n ($n < 32$)	equivalent to	rlwinm rA,rS,0,n,31
clrrwi rA,rS,n ($n < 32$)	equivalent to	rlwinm rA,rS,0,0,31 – n
clrlslwi rA,rS,b,n ($n < 32$)	equivalent to	rlwinm rA,rS,n,b – n,31 – n

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				M

rlwnm_x

rlwnm_x

Rotate Left Word then AND with Mask (x'5C00 0000')

rlwnm **rA,rS,rB,MB,ME** (**Rc** = '0')
rlwnm. **rA,rS,rB,MB,ME** (**Rc** = '1')

23	S	A	B	MB	ME	Rc
0	5 6	10 11	15 16	20 21	25 26	30 31

```

n ← rB[27-31]
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m

```

The contents of **rS** are rotated left the number of bits specified by the low-order 5 bits of **rB**. A mask is generated having '1' bits from bit **MB** through bit **ME** and '0' bits elsewhere. The rotated data is ANDed with the generated mask, and the result is placed into **rA**.

The **rlwnm** instruction can be used to extract and to rotate bit fields using one of the following methods:

- To extract an *n*-bit field, that starts at variable bit position *b* in **rS**, right-justified into **rA** (clearing the remaining 32 – *n* bits of **rA**) set:
rB (low-order 5 bits) = *b* + *n*
MB = 32 – *n*
ME = 31
- To extract an *n*-bit field, that starts at variable bit position *b* in **rS**, left-justified into **rA** (clearing the remaining 32 – *n* bits of **rA**), set:
rB (low-order 5 bits) = *b*
MB = '0'
ME = *n* – 1
- To rotate the contents of a register left (or right) by *n* bits, set:
rB (low-order 5 bits) = *n* (32 – *n*)
MB = '0'
ME = 31

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if **Rc** = '1')

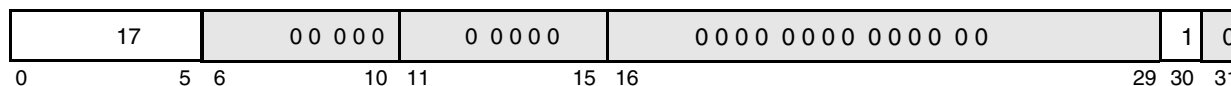
Simplified mnemonics:

rotlw **rA,rS,rB** equivalent to **rlwnm** **rA,rS,rB,0,31**

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				M

SC

System Call (x'4400 0002')

SC☐ Reserved

In the PowerPC UISA, the **sc** instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service.

This instruction is context synchronizing, as described in the context synchronizing instructions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- Dependent on the system service

In the PowerPC OEA, the **sc** instruction does the following:

```

SRR0 ← iea CIA + 4
SRR1[1-4, 10-15] ← 0
SRR1[0,5-9, 16-23, 25-27, 30-31] ← MSR[0,5-9, 16-23, 25-27, 30-31]
MSR ← new_value (information follows)
NIA ← iea base_ea + 0xC00 (information follows)

```

The EA of the instruction following the **sc** instruction is placed into SRR0. MSR bits [0, 5:9, 16:23, 25:27, 30:31] are placed into the corresponding bits of SRR1, and SRR1 bits [1:4, 10:15] are set to undefined values.

Note: An implementation can define additional MSR bits, and in this case, can also cause them to be saved to SRR1 from MSR on an exception and restored to MSR from SRR1 on an **rfi**. A system call exception is then generated. The exception causes the MSR to be altered as described in the exception definitions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

The exception causes the next instruction to be fetched from offset x'C00' from the physical base address determined by the new setting of MSR[IP].

Other registers altered:

- SRR0
- SRR1
- MSR

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA/OEA				SC

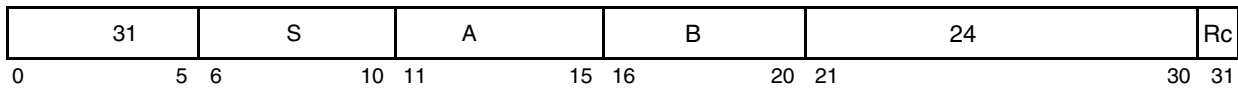
slw_x

Shift Left Word (x'7C00 0030')

slw_x

slw rA,rS,rB (Rc = '0')

slw. rA,rS,rB (Rc = '1')



```

n ← rB[27-31]
r ← ROTL(rS , n)
if rB[26] = 0 then
    m ← MASK(0 , 31 - n)
else
    m ← (32)0
rA ← r & m

```

The contents of rS are shifted left the number of bits specified by the low-order 5 bits of rB. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into rA. However, shift amounts from 32 - 63 give a '0' result.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

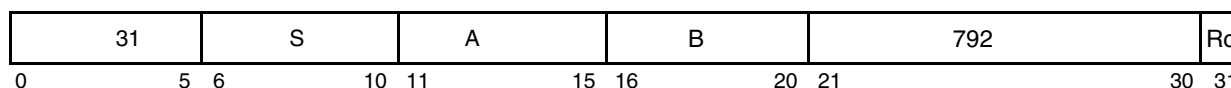
sraw_x

Shift Right Algebraic Word (x'7C00 0630')

sraw_x

sraw **rA,rS,rB** (Rc = '0')

sraw. **rA,rS,rB** (Rc = '1')



```

n ← rB[27-31]
r ← ROTL(rS, 32-n)
if rB[26] = 0 then
    m ← MASK(n, 31)
else
    m ← (32)0
S ← rS(0)
rA ← r & m | (32)S & ¬ m
XER[CA] ← S & ((r & ¬ m[0-31]) ≠ 0)

```

The contents of **rS** are shifted right the number of bits specified by the low-order 5 bits of **rB** (shift amounts of 0 - 31). Bits shifted out of position 31 are lost. Bit 0 of **rS** is replicated to fill the vacated positions on the left. The 32-bit result is placed into **rA**. **XER[CA]** is set if **rS** contains a negative number and any '1' bits are shifted out of position 31; otherwise **XER[CA]** is cleared. A shift amount of '0' causes **rA** to receive the 32 bits of **rS**, and **XER[CA]** to be cleared. However, shift amounts of 32 - 63 give a result of 32 sign bits, and cause **XER[CA]** to receive the sign bit of **rS**.

Note: The **sraw** instruction, followed by **addze**, can be used to divide quickly by 2^n .

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')
- XER
Affected: CA

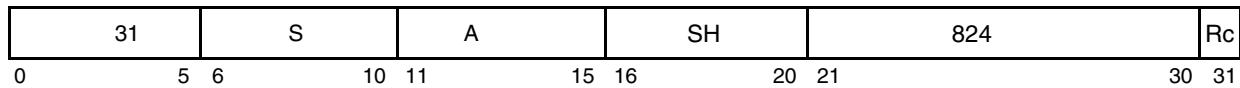
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

srawix

Shift Right Algebraic Word Immediate (x'7C00 0670')

srawix

srawi **rA,rS,SH** (Rc = '0')
srawi. **rA,rS,SH** (Rc = '1')



```

n ← SH
r ← ROTL(rS, 32-n)
m ← MASK(n, 31)
S ← rS(0)
rA ← r & m | (32)S & ¬ m
XER[CA] ← S(0) & ((r & ¬ m) ≠ 0)

```

The contents of **rS** are shifted right **SH** bits. Bits shifted out of position 31 are lost. Bit 0 of **rS** is replicated to fill the vacated positions on the left. The result is placed into **rA**. **XER[CA]** is set if the 32 bits of **rS** contain a negative number and any '1' bits are shifted out of position 31; otherwise, **XER[CA]** is cleared. A shift amount of '0' causes **rA** to receive the value of **rS**, and **XER[CA]** to be cleared.

Note: The **srawi** instruction, followed by **addze**, can be used to divide quickly by 2^n .

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')
- XER
Affected: CA

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

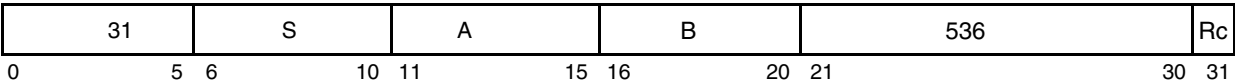
srw_x

Shift Right Word (x'7C00 0430')

srw_x

srw rA,rS,rB (Rc = '0')

srw. rA,rS,rB (Rc = '1')



```
n ← rB[27-31]
r ← ROTL(rS, 32-n)
if rB[26] = 0 then
    m ← MASK(n , 31)
else
    m ← (32)0
rA ← r & m
```

The contents of rS are shifted right the number of bits specified by the low-order 5 bits of rB (shift amounts of 0 - 31). Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into rA. However, shift amounts of 32 - 63 give a zero result.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

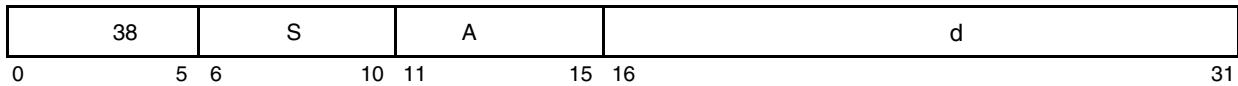
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stb

Store Byte (x'9800 0000')

stb

stb $rS, d(rA)$



```

if  $rA = 0$  then
     $b \leftarrow 0$ 
else
     $b \leftarrow (rA)$ 
 $EA \leftarrow b + EXTS(d)$ 
 $MEM(EA, 1) \leftarrow rS[24-31]$ 

```

EA is the sum $(rA \ll 0) + d$. The contents of the low-order 8 bits of rS are stored into the byte in memory addressed by EA.

Other registers altered:

- None

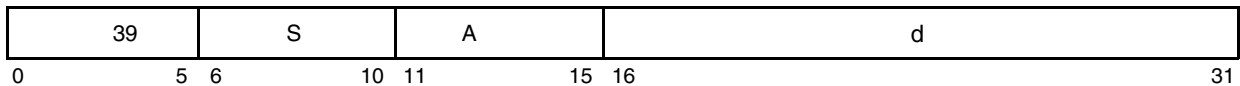
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

stbu

Store Byte with Update (x'9C00 0000')

stbu

stbu **rS,d(rA)**



```
EA ← (rA) + EXTS(d)
MEM(EA, 1) ← rS[24-31]
rA ← EA
```

EA is the sum (rA) + d. The contents of the low-order 8 bits of rS are stored into the byte in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

stbux

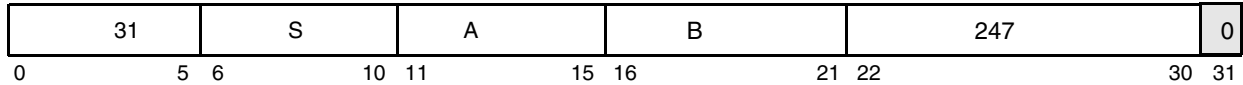
Store Byte with Update Indexed (x'7C00 01EE')

stbux

stbux

rS,rA,rB

 Reserved



$$EA \leftarrow (rA) + (rB)$$

$$MEM(EA, 1) \leftarrow rS[24-31]$$

$$rA \leftarrow EA$$

EA is the sum (rA) + (rB). The contents of the low-order 8 bits of rS are stored into the byte in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

- None

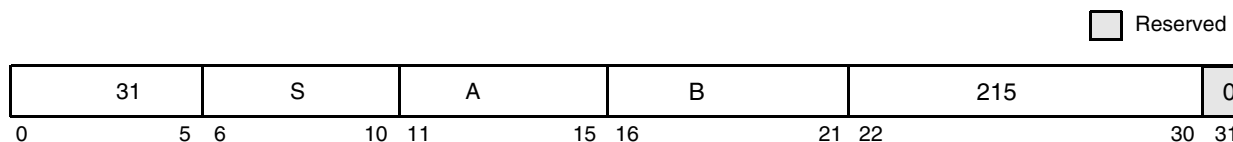
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stbx

Store Byte Indexed (x'7C00 01AE')

stbx r_S, r_A, r_B

stbx



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 1) ← rS[24-31]

```

EA is the sum ($rA[0] + rB$). The contents of the low-order 8 bits of rS are stored into the byte in memory addressed by EA.

Other registers altered:

- None

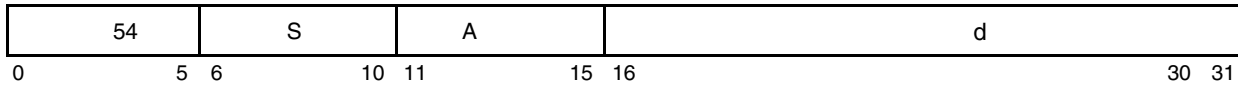
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stfd

Store Floating-Point Double (x'D800 0000')

stfd

stfd **frS,d(rA)**



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 8) ← (frS)

```

EA is the sum (**rA**l0) + d.

The contents of register **frS** are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

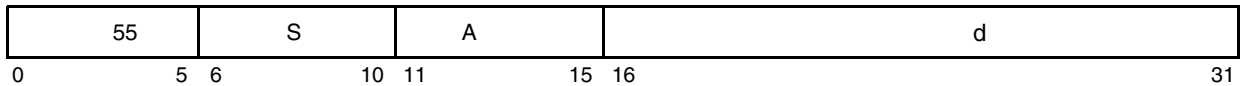
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

stfdu

stfdu

Store Floating-Point Double with Update (x'DC00 0000')

stfdu **frS,d(rA)**



```
EA ← (rA) + EXTS(d)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (rA) + d.

The contents of register frS are stored into the doubleword in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

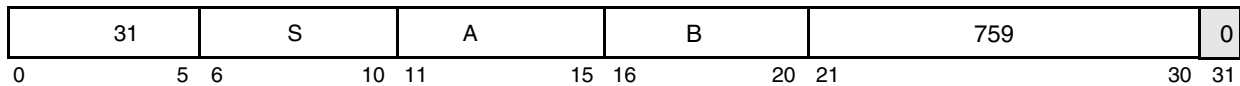
stfdux

stfdux

Store Floating-Point Double with Update Indexed (x'7C00 05EE')

stfdux **frS,rA,rB**

 Reserved



$EA \leftarrow (rA) + (rB)$
 $MEM(EA, 8) \leftarrow (frS)$
 $rA \leftarrow EA$

EA is the sum $(rA) + (rB)$.

The contents of register **frS** are stored into the doubleword in memory addressed by EA.

EA is placed into **rA**.

If **rA** = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

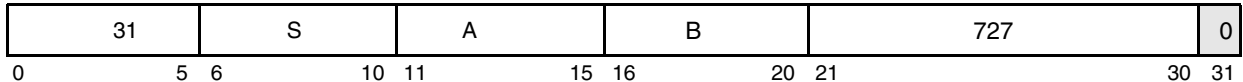
stfdx

stfdx

Store Floating-Point Double Indexed (x'7C00 05AE')

stfdx **frS,rA,rB**

 Reserved



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 8) ← (frS)
```

EA is the sum (**rA**l0) + **rB**.

The contents of register **frS** are stored into the doubleword in memory addressed by EA.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stfiwx

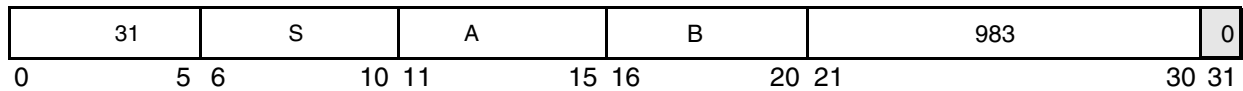
stfiwx

Store Floating-Point as Integer Word Indexed (x'7C00 07AE')

stfiwx

frS,rA,rB

Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← frS[32-63]

```

EA is the sum (rA|0) + (rB).

The contents of the low-order 32 bits of register **frS** are stored, without conversion, into the word in memory addressed by EA.

This instruction when preceded by the floating-point convert to integer word (**fctiwx**) or floating-point convert to integer word with round toward zero (**fctiwzx**) stores the 32-bit integer value of a double-precision floating-point number. (See *fctiwx* on page 317 and *fctiwzx* on page 318.)

Do not attempt to use this instruction to store the ps1 value for paired-single floating-point operands, the stored value is undefined.

If the content of register **frS** is a double-precision floating-point number, the low-order 32 bits of the 52-bit mantissa are stored. (Without the exponent, this could be a meaningless value.)

If the contents of register **frS** were produced, either directly or indirectly, by an **lfs** instruction, a single-precision arithmetic instruction, or **frsp**, then the value stored is the low-order 32 bits of the 52-bit mantissa of the double-precision number. (All single-precision floating-point numbers are maintained in double-precision format in the floating-point register file.)

When HID2[PSE] = '1', the input operand in **frS** must be the result of an **fctiw** or **fctiwz** instruction. Otherwise, the result is undefined.

Other registers altered:

- None

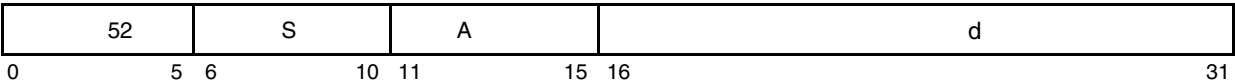
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA			YES	X

stfs

Store Floating-Point Single (x'D000 0000')

stfs

stfs **frS,d(rA)**



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (rA|0) + d.

The contents of register **frS** are converted to single-precision and stored into the word in memory addressed by EA. For a description of floating-point store conversions, see the floating-point store instructions section in the the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

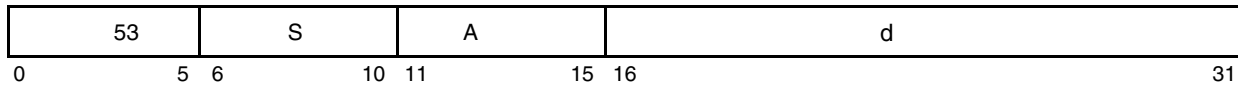
stfsu

stfsu

Store Floating-Point Single with Update (x'D400 0000')

stfsu

frS,d(rA)



```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (rA) + d.

The contents of **frS** are converted to single-precision and stored into the word in memory addressed by EA. For a description of floating-point store conversions, see the floating-point store instructions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

EA is placed into **rA**.

If **rA** = 0, the instruction form is invalid.

Other registers altered:

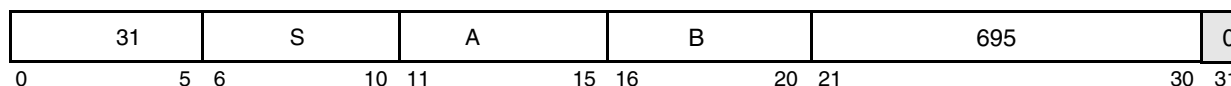
- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

stfsux

stfsux

Store Floating-Point Single with Update Indexed (x'7C00 056E')

stfsux**frS,rA,rB** Reserved

$$EA \leftarrow (rA) + (rB)$$

$$MEM(EA, 4) \leftarrow SINGLE(frS)$$

$$rA \leftarrow EA$$

EA is the sum $(rA) + (rB)$.

The contents of **frS** are converted to single-precision and stored into the word in memory addressed by EA. For a description of floating-point store conversions, see the floating-point store instructions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

EA is placed into **rA**.

If **rA** = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stfsx

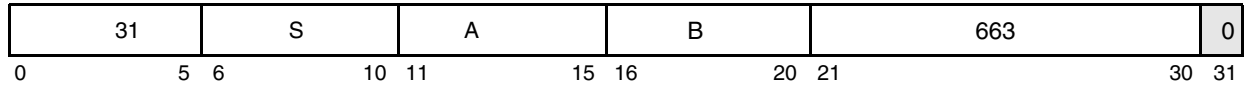
Store Floating-Point Single Indexed (x'7C00 052E')

stfsx

stfsx

frS,rA,rB

Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← SINGLE(frS)

```

EA is the sum (**rA**l0) + (**rB**).

The contents of register **frS** are converted to single-precision and stored into the word in memory addressed by EA. For a description of floating-point store conversions, see the floating-point store instructions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

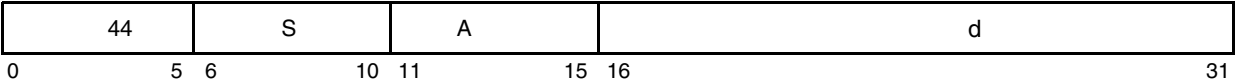


sth

Store Halfword (x'B000 0000')

sth

sth **rS,d(rA)**



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 2) ← rS[16-31]
```

EA is the sum (rA|0) + d. The contents of the low-order 16 bits of rS are stored into the halfword in memory addressed by EA.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

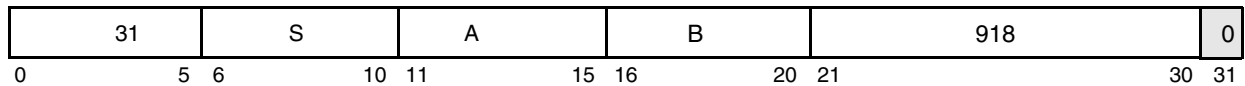
sthbrx

Store Halfword Byte-Reverse Indexed (x'7C00 072C')

sthbrx

sthbrx **rS,rA,rB**

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[24-31] || rS[16-23]

```

EA is the sum (rA|0) + (rB). The contents of the low-order 8 bits (24:31) of rS are stored into bits 0:7 of the halfword in memory addressed by EA. The contents of the subsequent low-order 8 bits (16:23) of rS are stored into bits 8:15 of the halfword in memory addressed by EA.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

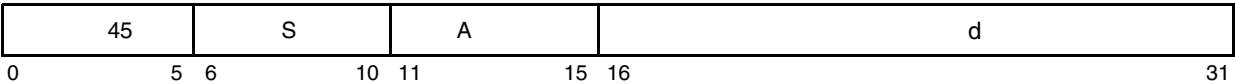


sth

Store Halfword with Update (x'B400 0000')

sth

sth **rS,d(rA)**



```
EA ← (rA) + EXTS(d)
MEM(EA, 2) ← rS[16-31]
rA ← EA
```

EA is the sum (rA) + d. The contents of the low-order 16 bits of rS are stored into the halfword in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

sthux

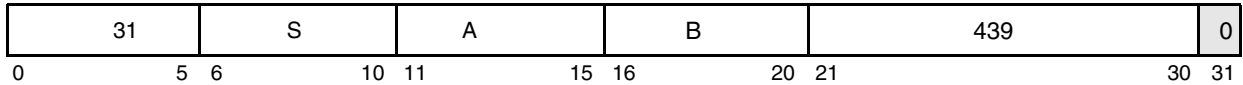
Store Halfword with Update Indexed (x'7C00 036E')

sthux

sthux

rS,rA,rB

 Reserved



$$EA \leftarrow (rA) + (rB)$$

$$MEM(EA, 2) \leftarrow rS[16-31]$$

$$rA \leftarrow EA$$

EA is the sum (rA) + (rB). The contents of the low-order 16 bits of rS are stored into the halfword in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

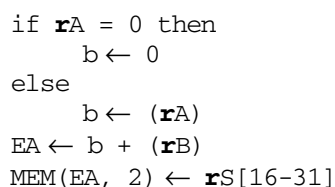
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

sthx

sthx **rS,rA,rB**



Other registers altered:

- None

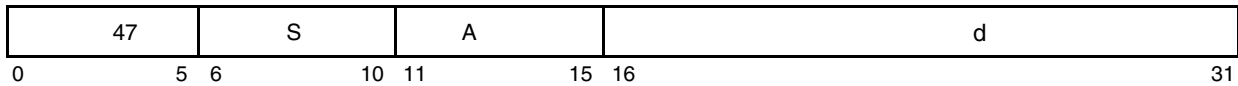
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stmw

Store Multiple Word (x'BC00 0000')

stmw

stmw $rS, d(rA)$



```

if  $rA = 0$  then
     $b \leftarrow 0$ 
else
     $b \leftarrow (rA)$ 
 $EA \leftarrow b + \text{EXTS}(d)$ 
 $r \leftarrow rS$ 
do while  $r \leq 31$ 
     $\text{MEM}(EA, 4) \leftarrow \text{GPR}(r)$ 
     $r \leftarrow r + 1$ 
     $EA \leftarrow EA + 4$ 

```

EA is the sum $(rA \ll 0) + d$.

$n = (32 - rS)$.

n consecutive words starting at EA are stored from the GPRs rS through $r31$. For example, if $rS = 30$, 2 words are stored.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see the DSI exception section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produce the same results.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

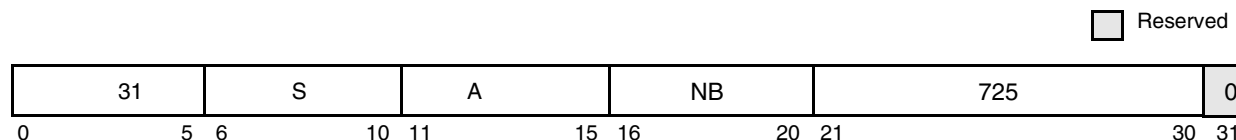
stswi

Store String Word Immediate (x'7C00 05AA')

stswi

rS,rA,NB

stswi



```

if rA = 0 then
    EA  $\leftarrow$  0
else
    EA  $\leftarrow$  (rA)
if NB = 0 then
    n  $\leftarrow$  32
else
    n  $\leftarrow$  NB
r  $\leftarrow$  rS - 1
i  $\leftarrow$  0
do while n > 0
    if i = 0 then
        r  $\leftarrow$  r + 1 (mod 32)
    MEM(EA, 1)  $\leftarrow$  GPR(r)[i, i + 7]
    i  $\leftarrow$  i + 8
    if i = 32 then
        i  $\leftarrow$  0
    EA  $\leftarrow$  EA + 1
    n  $\leftarrow$  n - 1

```

EA is (rAl0). Let $n = \text{NB}$ if $\text{NB} \neq 0$, $n = 32$ if $\text{NB} = 0$; n is the number of bytes to store. Let $nr = \text{CEIL}(n \div 4)$; nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs rS through $rS + nr - 1$. Bytes are stored left to right from each register. The sequence of registers wraps around through $r0$ if required.

Under certain conditions (for example, segment boundary crossing), the data alignment exception handler can be invoked. For additional information about data alignment exceptions, see the DSI exception section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produce the same results.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stswx

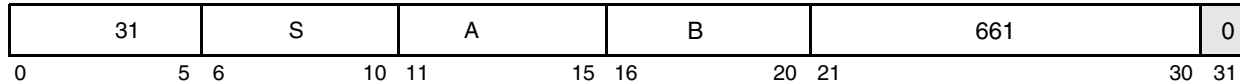
Store String Word Indexed (x'7C00 052A')

stswx

stswx

rS,rA,rB

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
n ← XER[25:31]
r ← rS - 1
i ← 0
do while n > 0
    if i = 0 then
        r ← r + 1 (mod 32)
        MEM(EA, 1) ← GPR(r) [i, i + 7]
        i ← i + 8
        if i = 32 then
            i ← 0
        EA ← EA + 1
        n ← n - 1

```

EA is the sum (rA|0) + (rB). Let $n = \text{XER}[25:31]$; n is the number of bytes to store.

Let $nr = \text{CEIL}(n \div 4)$; nr is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs rS through rS + nr - 1. Bytes are stored left to right from each register. The sequence of registers wraps around through r0 if required. If $n = 0$, no bytes are stored.

Under certain conditions (for example, segment boundary crossing), the data alignment exception handler can be invoked. For additional information about data alignment exceptions, see the DSI Exception section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Note: In some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual store instructions that produces the same results.

Other registers altered:

- None

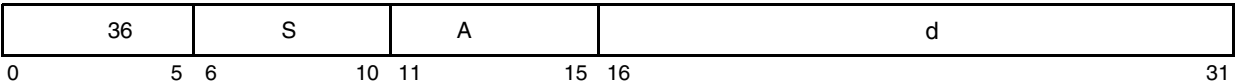
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stw

Store Word (x'9000 0000')

stw

stw **rS,d(rA)**



```
if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← rS
```

EA is the sum (**rA**l0) + d. The contents of **rS** are stored into the word in memory addressed by EA.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

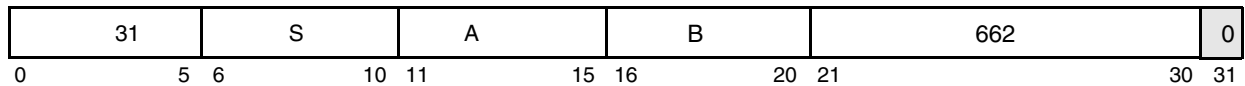
stwbrx

Store Word Byte-Reverse Indexed (x'7C00 052C')

stwbrx

stwbrx rS,rA,rB

Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS[24-31] || rS[16-23] || rS[8-15] || rS[0-7]

```

EA is the sum (rA|0) + (rB). The contents of the low-order 8 bits (24:31) of rS are stored into bits 0:7 of the word in memory addressed by EA. The contents of the subsequent 8 low-order bits (16:23) of rS are stored into bits 8:15 of the word in memory addressed by EA. The contents of the subsequent 8 low-order bits (8:15) of rS are stored into bits 16:23 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits (0:7) of rS are stored into bits 24:31 of the word in memory addressed by EA.

Other registers altered:

- None

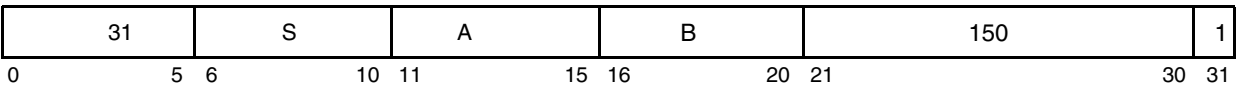
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stwcx.

stwcx.

Store Word Conditional Indexed (x'7C00 012D')

stwcx. **rS,rA,rB**



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
if RESERVE then
    MEM(EA, 4) ← rS
    CR0 ← 0b00 || 0b1 || XER[SO]
    RESERVE ← 0
else
    CR0 ← 0b00 || 0b0 || XER[SO]
    
```

EA is the sum (rA|0) + (rB). If the reserved bit is set, the **stwcx.** instruction stores **rS** to effective address (rA + rB), clears the reserved bit, and sets CR0[EQ]. If the reserved bit is not set, the **stwcx.** instruction does not do a store; it leaves the reserved bit cleared and clears CR0[EQ]. Software must look at CR0[EQ] to verify if the **stwcx.** instruction was successful.

The reserved bit is set by the **lwarx** instruction. The reserved bit is cleared by any **stwcx.** instruction to any address, and also by snooping logic if it detects that another processor does a write or invalidate to the block indicated in the reservation buffer when reserved is set.

EA must be a multiple of four. If it is not, either the system alignment exception handler is invoked, or the results are boundedly undefined. For additional information about alignment and DSI exceptions, see the DSI exception section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by the load and reserve and store conditional instructions should be controlled by a system library program.

Because the hardware does not compare reservation addresses when executing the **stwcx.** instruction, operating systems software must reset the reservation if an exception or other type of interrupt occurs to ensure atomic memory references of **lwarx** and **stwcx.** pairs.

Other registers altered:

- CR0 field is set to reflect whether the store operation was performed as follows:
CR0[LT, GT, EQ, SO] = 0b00 || store_performed || XER[SO]
- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO

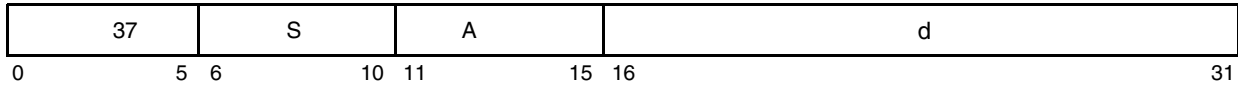
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stwu

Store Word with Update (x'9400 0000')

stwu

stwu **rS,d(rA)**



$$EA \leftarrow (rA) + \text{EXTS}(d)$$

$$\text{MEM}(EA, 4) \leftarrow rS$$

$$rA \leftarrow EA$$

EA is the sum $(rA) + d$. The contents of rS are stored into the word in memory addressed by EA.

EA is placed into rA .

If $rA = 0$, the instruction form is invalid.

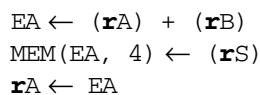
Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

stwux

stwux **rS,rA,rB**



- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

stwx

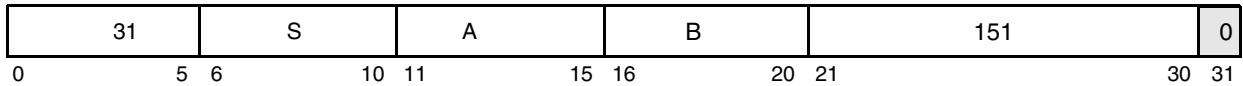
Store Word Indexed (x'7C00 012E')

stwx

stwx

rS,rA,rB

 Reserved



```

if rA = 0 then
    b ← 0
else
    b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← (rS)

```

EA is the sum (rA|0) + (rB). The contents of rS are stored into the word in memory addressed by EA.

Other registers altered:

- None

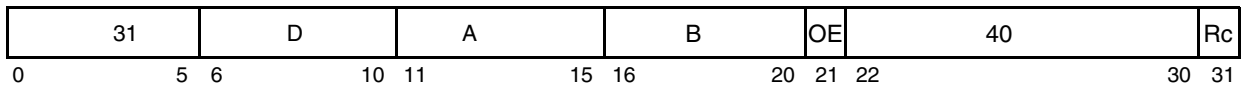
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

subf_x

Subtract From (x'7C00 0050')

subf_x

subf	rD,rA,rB	(OE = '0' Rc = '0')
subf.	rD,rA,rB	(OE = '0' Rc = '1')
subfo	rD,rA,rB	(OE = '1' Rc = '0')
subfo.	rD,rA,rB	(OE = '1' Rc = '1')



$$rD \leftarrow \neg (rA) + (rB) + 1$$

The sum $\neg (rA) + (rB) + 1$ is placed into **rD** [equivalent to $(rB)--(rA)$].

The **subf** instruction is preferred for subtraction because it sets few status bits.

Other registers altered:

- Condition Register (CR0 field):
Affected: LT, GT, EQ, SO (if Rc = '1')
- XER:
Affected: SO, OV (if OE = '1')

Simplified mnemonics:

sub **rD,rA,rB** equivalent to **subf** **rD,rB,rA**

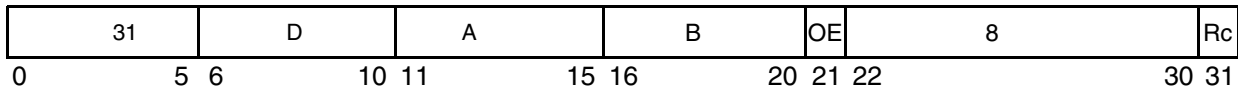
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

subfc_x

Subtract from Carrying (x'7C00 0010')

subfc_x

subfc	rD,rA,rB	(OE = '0' Rc = '0')
subfc.	rD,rA,rB	(OE = '0' Rc = '1')
subfco	rD,rA,rB	(OE = '1' Rc = '0')
subfco.	rD,rA,rB	(OE = '1' Rc = '1')



$$rD \leftarrow \neg (rA) + (rB) + 1$$

The sum $\neg (rA) + (rB) + 1$ is placed into **rD** [equivalent to $(rB) - (rA)$].

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

Note: The setting of the affected bits in the XER reflects overflow of the 32-bit results. For more information, see the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Simplified mnemonics:

subc **rD,rA,rB** equivalent to **subfc** **rD,rB,rA**

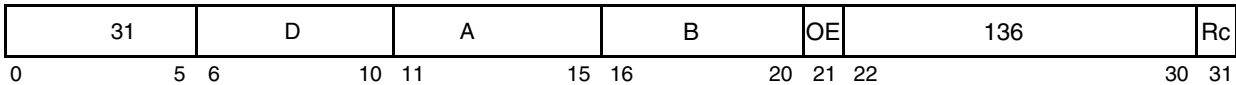
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

subfe_x

Subtract from Extended (x'7C00 0110')

subfe	rD,rA,rB	(OE = '0' Rc = '0')
subfe.	rD,rA,rB	(OE = '0' Rc = '1')
subfeo	rD,rA,rB	(OE = '1' Rc = '0')
subfeo.	rD,rA,rB	(OE = '1' Rc = '1')

subfe_x



$$rD \leftarrow \neg (rA) + (rB) + XER[CA]$$

The sum $\neg (rA) + (rB) + XER[CA]$ is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs. See the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual for setting of affected bits.

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

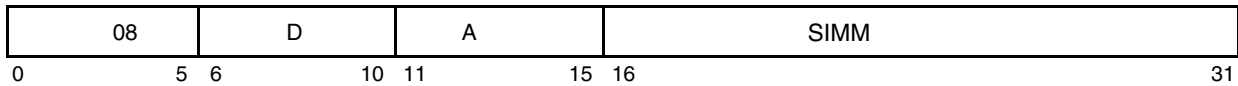


subfic

Subtract from Immediate Carrying (x'2000 0000')

subfic

subfic rD,rA,SIMM



$$rD \leftarrow \neg (rA) + \text{EXTS}(SIMM) + 1$$

The sum $\neg (rA) + \text{EXTS}(SIMM) + 1$ is placed into rD; equivalent to $\text{EXTS}(SIMM) - (rA)$.

Other registers altered:

- XER
Affected: CA

Note: See the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual for more information.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

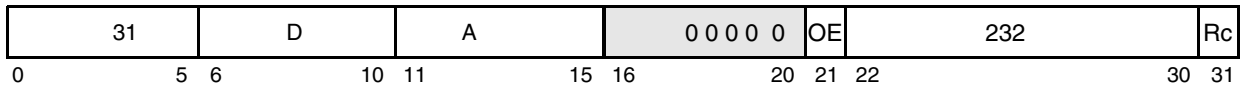
subfme_x

subfme_x

Subtract from Minus One Extended (x'7C00 01D0')

subfme	rD,rA	(OE = '0' Rc = '0')
subfme.	rD,rA	(OE = '0' Rc = '1')
subfmeo	rD,rA	(OE = '1' Rc = '0')
subfmeo.	rD,rA	(OE = '1' Rc = '1')

Reserved



$$rD \leftarrow \neg (rA) + XER[CA] - 1$$

The sum $\neg (rA) + XER[CA] + (32)1$ is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see the operand conventions section in the *PowerPC Microprocessor Family: The Programming Environments* manual).

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

subfze_x

Subtract from Zero Extended (x'7C00 0190')

subfze_x

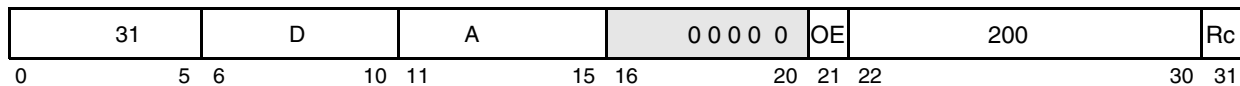
subfze rD,rA (OE = '0' Rc = '0')

subfze. rD,rA (OE = '0' Rc = '1')

subfzeo rD,rA (OE = '1' Rc = '0')

subfzeo. rD,rA (OE = '1' Rc = '1')

 Reserved



$$rD \leftarrow \neg (rA) + XER[CA]$$

The sum $\neg (rA) + XER[CA]$ is placed into rD.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

Note: CR0 field might not reflect the infinitely precise result if overflow occurs (see XER).

- XER
Affected: CA
Affected: SO, OV (if OE = '1')

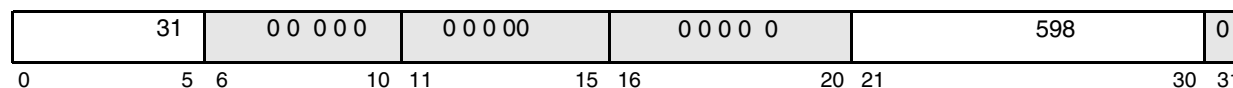
Note: See the operand conventions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				XO

sync

Synchronize (x'7C00 04AC')

sync

☐ Reserved

The **sync** instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a **sync** instruction ensures that all instructions preceding the **sync** instruction appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated by the processor until after the **sync** instruction completes. When the **sync** instruction completes, all external accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory. For more information about how the **sync** instruction affects the VEA, see the cache model and memory coherency section in the *PowerPC Microprocessor Family: The Programming Environments* manual.

Multiprocessor implementations also send a **sync** address-only broadcast that is useful in some designs. For example, if a design has an external buffer that reorders loads and stores for better bus efficiency, the **sync** broadcast signals to that buffer that previous loads/stores must be completed before any following loads/stores.

The **sync** instruction can be used to ensure that the results of all stores into a data structure, caused by store instructions executed in a critical section of a program, are detected by other processors before the data structure is detected as unlocked.

The functions performed by the **sync** instruction typically take a significant amount of time to complete, so that indiscriminate use of this instruction can adversely affect performance. In addition, the time required to execute a **sync** instruction can vary from one execution to another.

The **eieio** instruction might be more appropriate than a **sync** instruction for many cases.

This instruction is execution synchronizing. For more information about execution synchronization, see the synchronizing instructions section of the *PowerPC Microprocessor Family: The Programming Environments* manual.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

tlbie

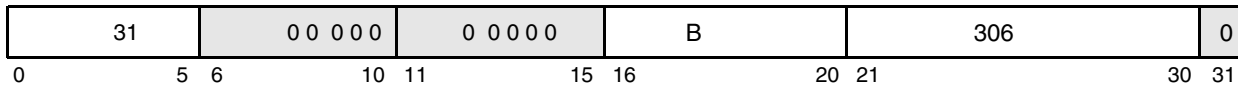
tlbie

Translation Lookaside Buffer Invalidate Entry (x'7C00 0264')

tlbie

rB

 Reserved



```
if 128 KB page then
    VPS ← rB[4-14]
else
    VPS ← rB[4-19]
```

```
Identify TLB entries corresponding to VPS
Each such TLB entry ← invalid
```

EA is the contents of **rB**. If the translation lookaside buffer (TLB) contains an entry corresponding to EA, that entry is made invalid (that is, removed from the TLB).

Multiprocessing implementations send a **tlbie** address-only broadcast over the address bus to instruct other processors to invalidate the same TLB entry in their TLBs.

The TLB search is done regardless of the settings of MSR[IR] and MSR[DR]. The search is done based on a portion of the logical page number within a segment, without reference to the SLB, segment table, or segment registers. All entries matching the search criteria are invalidated.

Block address translation for EA, if any, is ignored. See the synchronization of memory accesses and referenced and changed bit updates and the page table updates sections in the *PowerPC Microprocessor Family: The Programming Environments* manual for other requirements associated with the use of this instruction.

This is a supervisor-level instruction and is optional in the PowerPC Architecture.

Other registers altered:

- None

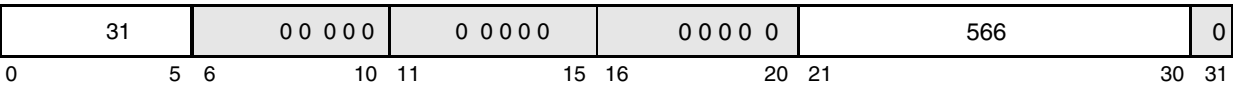
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	YES		YES	X

tlbsync

TLB Synchronize (x'7C00 046C')

tlbsync

 Reserved



If an implementation sends a broadcast for **tlbie**, it also sends a broadcast for **tlbsync**. Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all other processors.

The operation performed by this instruction is treated as a caching-inhibited and guarded data access with respect to the ordering done by **eieio**.

See the synchronization of memory accesses and referenced and changed bit updates and the page table updates sections in the *PowerPC Microprocessor Family: The Programming Environments* manual for other requirements associated with the use of this instruction.

Note: Espresso implements the effect of a **tlbsync** broadcast using internal signalling among the cores. When a TLBIE operation is broadcast from one core to the others, those cores each assert a **tlbsync** signal back to the first core indicating that a TLBIE operation is pending. The **tlbsync** instruction is not completed unless the **tlbsync** signals are all negated.

This instruction is supervisor-level and is optional in the PowerPC Architecture.

Other registers altered:

- None

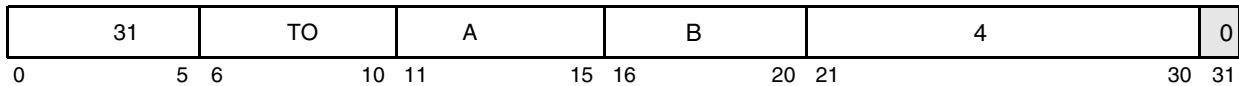
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
OEA	YES		YES	X

tw

tw

Trap Word (x'7C00 0008')

tw TO,rA,rB

 Reserved


```

a ← EXTS(rA)
b ← EXTS(rB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP

```

The contents of **rA** are compared arithmetically with the contents **rB** for TO[0:2]. The contents of **rA** are compared logically with the contents **rB** for TO[3:4]. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

tweq	rA,rB	equivalent to	tw	4,rA,rB
twlge	rA,rB	equivalent to	tw	5,rA,rB
trap		equivalent to	tw	31,0,0

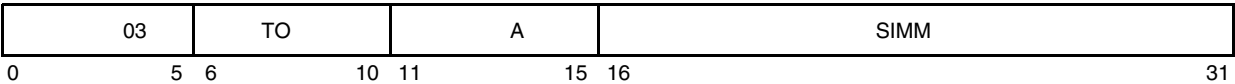
PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

twi

twi

Trap Word Immediate (x'0C00 0000')

twi TO,rA,SIMM



```
a ← EXTS(rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

The contents of **rA** are compared arithmetically with the sign-extended value of the SIMM field for TO[0:2]. The contents of **rA** are compared logically with the sign-extended value of the SIMM field for TO[3:4]. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

twgti	rA,value	equivalent to	twi	8,rA,value
twllel	rA,value	equivalent to	twi	6,rA,value

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D



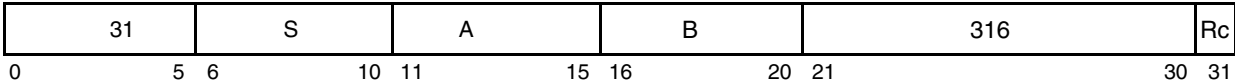
xor_x
XOR (x'7C00 0278')

xor_x

xor
xor.

rA,rS,rB
rA,rS,rB

(Rc = '0')
(Rc = '1')



rA ← (**rS**) ⊕ (**rB**)

The contents of **rS** are XORed with the contents of **rB**, and the result is placed into **rA**.

Other registers altered:

- Condition Register (CR0 field)
Affected: LT, GT, EQ, SO (if Rc = '1')

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				X

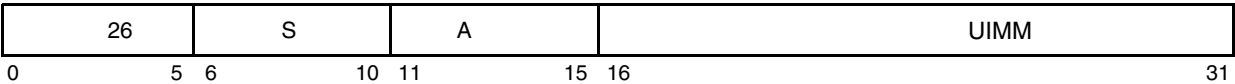


xori

XOR Immediate (x'6800 0000')

xori

xori **rA,rS,UIMM**



$$rA \leftarrow (rS) \oplus ((16)0 \parallel UIMM)$$

The contents of **rS** are XORed with 0x0000 || UIMM, and the result is placed into **rA**.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D

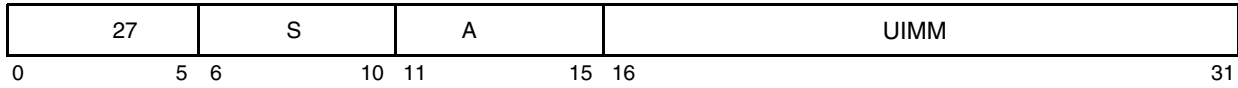
xoris

XOR Immediate Shifted (x'6C00 0000')

xoris

xoris

rA,rS,UIMM



$$rA \leftarrow (rS) \oplus (UIMM \parallel (16)0)$$

The contents of **rS** are XORed with UIMM || 0x0000, and the result is placed into **rA**.

Other registers altered:

- None

PowerPC Architecture Level	Supervisor Level	Espresso Specific	PowerPC Optional	Form
UISA				D



Appendix A. Espresso Instruction Set

A.1 Instructions Sorted by Opcode

Table A-1. lists the instructions defined in the PowerPC Architecture in numeric order by opcode.

Key:

 Reserved bits

Table A-1. Complete Instruction List Sorted by Opcode (Page 1 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
twi	0 0 0 0 1 1	TO			A			SIMM																				
ps_cmpu0	000100	crfD	00		A			B			0000000000															0		
psq_lx	000100	D			A			B			w	i		000110					0									
psq_stx	000100	S			A			B			w	i		000111					0									
ps_sum0	000100	D			A			B			C			01010					Rc									
ps_sum1	000100	D			A			B			C			01011					Rc									
ps_muls0	000100	D			A			00000			C			01100					Rc									
ps_muls1	000100	D			A			00000			C			01101					Rc									
ps_madds0	000100	D			A			B			C			01110					Rc									
ps_madds1	000100	D			A			B			C			01111					Rc									
ps_div	000100	D			A			B			00000			10010					Rc									
ps_sub	000100	D			A			B			00000			10100					Rc									
ps_add	000100	D			A			B			00000			10101					Rc									
ps_sel	000100	D			A			B			C			10111					Rc									
ps_res	000100	D			00000			B			00000			11000					Rc									
ps_mul	000100	D			A			00000			C			11001					Rc									
ps_rsqrte	000100	D			00000			B			00000			11010					Rc									
ps_msub	000100	D			A			B			C			11100					Rc									
ps_madd	000100	D			A			B			C			11101					Rc									
ps_nmsub	000100	D			A			B			C			11110					Rc									
ps_nmadd	000100	D			A			B			C			11111					Rc									
ps_cmpo0	000100	crfD	0 0		A			B			0000100000															0		
psq_lux	000100	D			A			B			w	i		100110					0									
psq_stux	000100	S			A			B			w	i		100111					0									
ps_neg	000100	D			00000			B			0000101000															Rc		
ps_cmpu1	000100	crfD	00		A			B			0001000000															0		
ps_mr	000100	D			00000			B			0001001000															Rc		
ps_cmpo1	000100	crfD	00		A			B			0001100000															0		

Table A-1. Complete Instruction List Sorted by Opcode (Page 2 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31										
ps_nabs	000100				D				00000					B								0010001000							Rc									
ps_abs	000100				D				00000					B								0100001000							Rc									
ps_merge00	000100				D				A					B								1000010000							Rc									
ps_merge01	000100				D				A					B								1000110000							Rc									
ps_merge10	000100				D				A					B								1001010101							Rc									
ps_merge11	000100				D				A					B								1001110000							Rc									
dcbz_l	000100				00000				A					B								1111110110							0									
mulli	0 0 0 1 1 1				D				A													SIMM																
subfic	0 0 1 0 0 0				D				A													SIMM																
cmpli	0 0 1 0 1 0				crfD		0	L		A												UIMM																
cmpi	0 0 1 0 1 1				crfD		0	L		A												SIMM																
addic	0 0 1 1 0 0				D				A													SIMM																
addic.	0 0 1 1 0 1				D				A													SIMM																
addi	0 0 1 1 1 0				D				A													SIMM																
addis	0 0 1 1 1 1				D				A													SIMM																
bcx	0 1 0 0 0 0				BO				BI													BD										AA	LK					
sc	0 1 0 0 0 1				0 0 0 0 0				0 0 0 0 0					0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																1	0							
bx	0 1 0 0 1 0				LI																										AA	LK						
mcrf	0 1 0 0 1 1				crfD		0 0		crfS		0 0		0 0 0 0 0								0 0 0 0 0 0 0 0 0 0										0							
bclrx	0 1 0 0 1 1				BO				BI					0 0 0 0 0							0 0 0 0 0 1 0 0 0 0										LK							
crnor	0 1 0 0 1 1				crbD				crbA				crbB								0 0 0 0 1 0 0 0 0 1										0							
rfi	0 1 0 0 1 1				0 0 0 0 0				0 0 0 0 0				0 0 0 0 0								0 0 0 0 1 1 0 0 1 0										0							
crandc	0 1 0 0 1 1				crbD				crbA				crbB								0 0 1 0 0 0 0 0 0 1										0							
isync	0 1 0 0 1 1				0 0 0 0 0				0 0 0 0 0				0 0 0 0 0								0 0 1 0 0 1 0 1 1 0										0							
crxor	0 1 0 0 1 1				crbD				crbA				crbB								0 0 1 1 0 0 0 0 0 1										0							
crnand	0 1 0 0 1 1				crbD				crbA				crbB								0 0 1 1 1 0 0 0 0 1										0							
crand	0 1 0 0 1 1				crbD				crbA				crbB								0 1 0 0 0 0 0 0 0 1										0							
creqv	0 1 0 0 1 1				crbD				crbA				crbB								0 1 0 0 1 0 0 0 0 1										0							
crorc	0 1 0 0 1 1				crbD				crbA				crbB								0 1 1 0 1 0 0 0 0 1										0							
cror	0 1 0 0 1 1				crbD				crbA				crbB								0 1 1 1 0 0 0 0 0 1										0							
bcctrx	0 1 0 0 1 1				BO				BI					0 0 0 0 0							1 0 0 0 0 1 0 0 0 0										LK							
rlwimix	0 1 0 1 0 0				S				A				SH								MB				ME				Rc									
rlwinmx	0 1 0 1 0 1				S				A				SH								MB				ME				Rc									
rlwnmx	0 1 0 1 1 1				S				A				B								MB				ME				Rc									
ori	0 1 1 0 0 0				S				A																				UIMM									
oris	0 1 1 0 0 1				S				A																				UIMM									
xori	0 1 1 0 1 0				S				A																				UIMM									



Advance

IBM Espresso RISC Microprocessor

Table A-1. Complete Instruction List Sorted by Opcode (Page 3 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
xoris	0 1 1 0 1 1	S			A			UIMM																				
andi.	0 1 1 1 0 0	S			A			UIMM																				
andis.	0 1 1 1 0 1	S			A			UIMM																				
cmp	0 1 1 1 1 1	crfD	0	L	A			B			0 0 0 0 0 0 0 0 0 0															0		
tw	0 1 1 1 1 1	TO			A			B			0 0 0 0 0 0 0 1 0 0															0		
subfcx	0 1 1 1 1 1	D			A			B			OE	0 0 0 0 0 0 1 0 0 0															Rc	
addcx	0 1 1 1 1 1	D			A			B			OE	0 0 0 0 0 0 1 0 1 0															Rc	
mulhwux	0 1 1 1 1 1	D			A			B			0	0 0 0 0 0 0 1 0 1 1															Rc	
mfcrr	0 1 1 1 1 1	D			0 0 0 0 0			0 0 0 0 0			0 0 0 0 0 1 0 0 1 1															0		
lwarx	0 1 1 1 1 1	D			A			B			0 0 0 0 0 1 0 1 0 0															0		
lwzx	0 1 1 1 1 1	D			A			B			0 0 0 0 0 1 0 1 1 1															0		
slwx	0 1 1 1 1 1	S			A			B			0 0 0 0 0 1 1 0 0 0															Rc		
cntlzwx	0 1 1 1 1 1	S			A			0 0 0 0 0			0 0 0 0 0 1 1 0 1 0															Rc		
andx	0 1 1 1 1 1	S			A			B			0 0 0 0 0 1 1 1 0 0															Rc		
cmpl	0 1 1 1 1 1	crfD	0	L	A			B			0 0 0 0 1 0 0 0 0 0															0		
subfx	0 1 1 1 1 1	D			A			B			OE	0 0 0 0 1 0 1 0 0 0															Rc	
dcbst	0 1 1 1 1 1	0 0 0 0 0			A			B			0 0 0 0 1 1 0 1 1 0															0		
lwzux	0 1 1 1 1 1	D			A			B			0 0 0 0 1 1 0 1 1 1															0		
andcx	0 1 1 1 1 1	S			A			B			0 0 0 0 1 1 1 1 0 0															Rc		
mulhwx	0 1 1 1 1 1	D			A			B			0	0 0 0 1 0 0 1 0 1 1															Rc	
mfmsr	0 1 1 1 1 1	D			0 0 0 0 0			0 0 0 0 0			0 0 0 1 0 1 0 0 1 1															0		
dcbf	0 1 1 1 1 1	0 0 0 0 0			A			B			0 0 0 1 0 1 0 1 1 0															0		
lbzx	0 1 1 1 1 1	D			A			B			0 0 0 1 0 1 0 1 1 1															0		
negx	0 1 1 1 1 1	D			A			0 0 0 0 0			OE	0 0 0 1 1 0 1 0 0 0															Rc	
lbzux	0 1 1 1 1 1	D			A			B			0 0 0 1 1 1 0 1 1 1															0		
norx	0 1 1 1 1 1	S			A			B			0 0 0 1 1 1 1 1 0 0															Rc		
subfex	0 1 1 1 1 1	D			A			B			OE	0 0 1 0 0 0 1 0 0 0															Rc	
addex	0 1 1 1 1 1	D			A			B			OE	0 0 1 0 0 0 1 0 1 0															Rc	
mtcrf	0 1 1 1 1 1	S			0	CRM			0	0 0 1 0 0 1 0 0 0 0															0			
mtmsr	0 1 1 1 1 1	S			0 0 0 0 0			0 0 0 0 0			0 0 1 0 0 1 0 0 1 0															0		
stwcx.	0 1 1 1 1 1	S			A			B			0 0 1 0 0 1 0 1 1 0															1		
stwx	0 1 1 1 1 1	S			A			B			0 0 1 0 0 1 0 1 1 1															0		
stwux	0 1 1 1 1 1	S			A			B			0 0 1 0 1 1 0 1 1 1															0		
subfzex	0 1 1 1 1 1	D			A			0 0 0 0 0			OE	0 0 1 1 0 0 1 0 0 0															Rc	
addzex	0 1 1 1 1 1	D			A			0 0 0 0 0			OE	0 0 1 1 0 0 1 0 1 0															Rc	
mtsr	0 1 1 1 1 1	S			0	SR			0 0 0 0 0			0 0 1 1 0 1 0 0 1 0															0	
stbx	0 1 1 1 1 1	S			A			B			0 0 1 1 0 1 0 1 1 1															0		

Table A-1. Complete Instruction List Sorted by Opcode (Page 4 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
subfmex	0 1 1 1 1 1	D			A			0 0 0 0 0			OE		0 0 1 1 1 1 0 1 0 0 0										Rc					
addmex	0 1 1 1 1 1	D			A			0 0 0 0 0			OE		0 0 1 1 1 1 0 1 0 1 0										Rc					
mullwx	0 1 1 1 1 1	D			A			B			OE		0 0 1 1 1 1 0 1 0 1 1										Rc					
mtsrin	0 1 1 1 1 1	S			0 0 0 0 0			B			0 0 1 1 1 1 1 0 0 1 0										0							
dcbtst	0 1 1 1 1 1	0 0 0 0 0			A			B			0 0 1 1 1 1 1 0 1 1 0										0							
stbux	0 1 1 1 1 1	S			A			B			0 0 1 1 1 1 1 0 1 1 1										0							
addx	0 1 1 1 1 1	D			A			B			OE		0 1 0 0 0 0 1 0 1 0										Rc					
dcbt	0 1 1 1 1 1	0 0 0 0 0			A			B			0 1 0 0 0 1 0 1 1 0										0							
lhzx	0 1 1 1 1 1	D			A			B			0 1 0 0 0 1 0 1 1 1										0							
eqvx	0 1 1 1 1 1	S			A			B			0 1 0 0 0 1 1 1 0 0										Rc							
tlbie	0 1 1 1 1 1	0 0 0 0 0			0 0 0 0 0			B			0 1 0 0 1 1 0 0 1 0										0							
eciwx	0 1 1 1 1 1	D			A			B			0 1 0 0 1 1 0 1 1 0										0							
lhzux	0 1 1 1 1 1	D			A			B			0 1 0 0 1 1 0 1 1 1										0							
xorx	0 1 1 1 1 1	S			A			B			0 1 0 0 1 1 1 1 0 0										Rc							
mfspir	0 1 1 1 1 1	D			spr										0 1 0 1 0 1 0 0 1 1										0			
lhax	0 1 1 1 1 1	D			A			B			0 1 0 1 0 1 0 1 1 1										0							
mftb	0 1 1 1 1 1	D			tbr										0 1 0 1 1 1 0 0 1 1										0			
lhaux	0 1 1 1 1 1	D			A			B			0 1 0 1 1 1 0 1 1 1										0							
sthx	0 1 1 1 1 1	S			A			B			0 1 1 0 0 1 0 1 1 1										0							
orcx	0 1 1 1 1 1	S			A			B			0 1 1 0 0 1 1 1 0 0										Rc							
ecowx	0 1 1 1 1 1	S			A			B			0 1 1 0 1 1 0 1 1 0										0							
sthux	0 1 1 1 1 1	S			A			B			0 1 1 0 1 1 0 1 1 1										0							
orx	0 1 1 1 1 1	S			A			B			0 1 1 0 1 1 1 1 0 0										Rc							
divwux	0 1 1 1 1 1	D			A			B			OE		0 1 1 1 0 0 1 0 1 1										Rc					
mtspr	0 1 1 1 1 1	S			spr										0 1 1 1 0 1 0 0 1 1										0			
dcbi	0 1 1 1 1 1	0 0 0 0 0			A			B			0 1 1 1 0 1 0 1 1 0										0							
nandx	0 1 1 1 1 1	S			A			B			0 1 1 1 0 1 1 1 0 0										Rc							
divwx	0 1 1 1 1 1	D			A			B			OE		0 1 1 1 1 0 1 0 1 1										Rc					
mcrxr	0 1 1 1 1 1	crfD		0 0		0 0 0 0 0			0 0 0 0 0			1 0 0 0 0 0 0 0 0 0										0						
lswx	0 1 1 1 1 1	D			A			B			1 0 0 0 0 1 0 1 0 1										0							
lwbrx	0 1 1 1 1 1	D			A			B			1 0 0 0 0 1 0 1 1 0										0							
lfsx	0 1 1 1 1 1	D			A			B			1 0 0 0 0 1 0 1 1 1										0							
srwx	0 1 1 1 1 1	S			A			B			1 0 0 0 0 1 1 0 0 0										Rc							
tlbsync	0 1 1 1 1 1	0 0 0 0 0			0 0 0 0 0			0 0 0 0 0			1 0 0 0 1 1 0 1 1 0										0							
lfsux	0 1 1 1 1 1	D			A			B			1 0 0 0 1 1 0 1 1 1										0							
mfsr	0 1 1 1 1 1	D			0		SR		0 0 0 0 0			1 0 0 1 0 1 0 0 1 1										0						
lswi	0 1 1 1 1 1	D			A			NB			1 0 0 1 0 1 0 1 0 1										0							



Table A-1. Complete Instruction List Sorted by Opcode (Page 5 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
sync	0 1 1 1 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					1 0 0 1 0 1 0 1 1 0										0
lfdx	0 1 1 1 1 1		D						A					B				1 0 0 1 0 1 0 1 1 1										0
lfdux	0 1 1 1 1 1		D						A					B				1 0 0 1 1 1 0 1 1 1										0
mfsrin	0 1 1 1 1 1		D					0 0 0 0 0						B				1 0 1 0 0 1 0 0 1 1										0
stswx	0 1 1 1 1 1		S						A					B				1 0 1 0 0 1 0 1 0 1										0
stwbrx	0 1 1 1 1 1		S						A					B				1 0 1 0 0 1 0 1 1 0										0
stfsx	0 1 1 1 1 1		S						A					B				1 0 1 0 0 1 0 1 1 1										0
stfsux	0 1 1 1 1 1		S						A					B				1 0 1 0 1 1 0 1 1 1										0
stswi	0 1 1 1 1 1		S						A					NB				1 0 1 1 0 1 0 1 0 1										0
stfdx	0 1 1 1 1 1		S						A					B				1 0 1 1 0 1 0 1 1 1										0
stfdux	0 1 1 1 1 1		S						A					B				1 0 1 1 1 1 0 1 1 1										0
lhbrx	0 1 1 1 1 1		D						A					B				1 1 0 0 0 1 0 1 1 0										0
srawx	0 1 1 1 1 1		S						A					B				1 1 0 0 0 1 1 0 0 0										Rc
srawix	0 1 1 1 1 1		S						A					SH				1 1 0 0 1 1 1 0 0 0										Rc
eieio	0 1 1 1 1 1	0 0 0 0 0						0 0 0 0 0					0 0 0 0 0					1 1 0 1 0 1 0 1 1 0										0
sthbrx	0 1 1 1 1 1		S						A					B				1 1 1 0 0 1 0 1 1 0										0
extshx	0 1 1 1 1 1		S						A					0 0 0 0 0				1 1 1 0 0 1 1 0 1 0										Rc
extsbx	0 1 1 1 1 1		S						A					0 0 0 0 0				1 1 1 0 1 1 1 0 1 0										Rc
icbi	0 1 1 1 1 1	0 0 0 0 0							A					B				1 1 1 1 0 1 0 1 1 0										0
stfiwx	0 1 1 1 1 1		S						A					B				1 1 1 1 0 1 0 1 1 1										0
dcbz	0 1 1 1 1 1	0 0 0 0 0							A					B				1 1 1 1 1 1 0 1 1 0										0
lwz	1 0 0 0 0 0		D						A									d										
lwzu	1 0 0 0 0 1		D						A									d										
lbz	1 0 0 0 1 0		D						A									d										
lbzu	1 0 0 0 1 1		D						A									d										
stw	1 0 0 1 0 0		S						A									d										
stwu	1 0 0 1 0 1		S						A									d										
stb	1 0 0 1 1 0		S						A									d										
stbu	1 0 0 1 1 1		S						A									d										
lhz	1 0 1 0 0 0		D						A									d										
lhzu	1 0 1 0 0 1		D						A									d										
lha	1 0 1 0 1 0		D						A									d										
lhau	1 0 1 0 1 1		D						A									d										
sth	1 0 1 1 0 0		S						A									d										
sthu	1 0 1 1 0 1		S						A									d										
lmw	1 0 1 1 1 0		D						A									d										
stmw	1 0 1 1 1 1		S						A									d										

Table A-1. Complete Instruction List Sorted by Opcode (Page 6 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
lfs	1 1 0 0 0 0	D			A			d																							
lfsu	1 1 0 0 0 1	D			A			d																							
lfd	1 1 0 0 1 0	D			A			d																							
lfdv	1 1 0 0 1 1	D			A			d																							
stfs	1 1 0 1 0 0	S			A			d																							
stfsu	1 1 0 1 0 1	S			A			d																							
stfd	1 1 0 1 1 0	S			A			d																							
stfdv	1 1 0 1 1 1	S			A			d																							
psq_l	111000	D			A			w	i		d																				
psq_lu	111001	D			A			w	i		d																				
fdvix	1 1 1 0 1 1	D			A			B			00000			10010			Rc														
fsubsx	1 1 1 0 1 1	D			A			B			00000			10100			Rc														
faddsx	1 1 1 0 1 1	D			A			B			00000			10101			Rc														
fresx	1 1 1 0 1 1	D			00000			B			00000			11000			Rc														
fmulx	1 1 1 0 1 1	D			A			00000			C			11001			Rc														
fmsubsx	1 1 1 0 1 1	D			A			B			C			11100			Rc														
fmaddsx	1 1 1 0 1 1	D			A			B			C			11101			Rc														
fnmsubsx	1 1 1 0 1 1	D			A			B			C			11110			Rc														
fnmaddsx	1 1 1 0 1 1	D			A			B			C			11111			Rc														
psq_st	111100	S			A			w	i		d																				
psq_stu	111101	S			A			w	i		d																				
fcmphu	1 1 1 1 1 1	crfD	00		A			B			0000000000															0					
frspx	1 1 1 1 1 1	D			00000			B			0000001100															Rc					
fctiw	1 1 1 1 1 1	D			00000			B			0000001110															Rc					
fctiwzx	1 1 1 1 1 1	D			00000			B			0000001111															Rc					
fdvix	1 1 1 1 1 1	D			A			B			00000			10010			Rc														
fsubx	1 1 1 1 1 1	D			A			B			00000			10100			Rc														
faddx	1 1 1 1 1 1	D			A			B			00000			10101			Rc														
fselx	1 1 1 1 1 1	D			A			B			C			10111			Rc														
fmulx	1 1 1 1 1 1	D			A			00000			C			11001			Rc														
frsqrtex	1 1 1 1 1 1	D			00000			B			00000			11010			Rc														
fmsubx	1 1 1 1 1 1	D			A			B			C			11100			Rc														
fmaddx	1 1 1 1 1 1	D			A			B			C			11101			Rc														
fnmsubx	1 1 1 1 1 1	D			A			B			C			11110			Rc														
fnmaddx	1 1 1 1 1 1	D			A			B			C			11111			Rc														
fcmppo	1 1 1 1 1 1	crfD	00		A			B			0000100000															0					
mtfsb1x	1 1 1 1 1 1	crbD			00000			00000			0000100110															Rc					



Table A-1. Complete Instruction List Sorted by Opcode (Page 7 of 7)

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31				
fnegx	1	1	1	1	1	1		D				0	0	0	0		B				0	0	0	0	1	0	1	0	0	Rc		
mcrfs	1	1	1	1	1	1		crfD		0	0		crfS		0	0		0	0	0	0	0	0	0	0	0	0	0	0	0		
mtfsb0x	1	1	1	1	1	1		crbD				0	0	0	0		0	0	0	0			0	0	0	1	0	0	1	1	Rc	
fmrx	1	1	1	1	1	1		D				0	0	0	0		B					0	0	0	1	0	0	1	0	0	Rc	
mtfsfix	1	1	1	1	1	1		crfD		0	0		0	0	0	0		IMM		0			0	0	1	0	0	0	0	1	1	Rc
fnabsx	1	1	1	1	1	1		D				0	0	0	0		B					0	0	1	0	0	0	1	0	0	Rc	
fabsx	1	1	1	1	1	1		D				0	0	0	0		B					0	1	0	0	0	0	1	0	0	Rc	
mffsx	1	1	1	1	1	1		D				0	0	0	0		0	0	0	0			1	0	0	1	0	0	0	1	1	Rc
mtfsfx	1	1	1	1	1	1		0				FM			0		B					1	0	1	1	0	0	0	1	1	1	Rc

A.2 Instructions Grouped by Functional Categories

Table A-2. through Table A-32. on page 512 list the Espresso instructions grouped by function.

Key:  Reserved bits

Table A-2. Integer Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
addx	31				D					A					B			OE					266					Rc
addcx	31				D					A					B			OE					10					Rc
addex	31				D					A					B			OE					138					Rc
addi	14				D					A																		SIMM
addic	12				D					A																		SIMM
addic.	13				D					A																		SIMM
addis	15				D					A																		SIMM
addmex	31				D					A					0	0	0	0	0	0		OE						Rc
addzex	31				D					A					0	0	0	0	0	0		OE						Rc
divwx	31				D					A					B				OE					491				Rc
divwux	31				D					A					B				OE					459				Rc
mulhwx	31				D					A					B			0						75				Rc
mulhwux	31				D					A					B			0						11				Rc
mulli	07				D					A																		SIMM
mullwx	31				D					A					B				OE					235				Rc
negx	31				D					A					0	0	0	0	0	0		OE						Rc
subfx	31				D					A					B				OE					40				Rc
subfcx	31				D					A					B				OE					8				Rc
subfex	31				D					A					B				OE					136				Rc
subficx	08				D					A																		SIMM
subfmex	31				D					A					0	0	0	0	0	0		OE						Rc
subfzex	31				D					A					0	0	0	0	0	0		OE						Rc

Table A-3. Integer Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
cmp	31		crfD		0	L		A						B								0						0
cmpi	11		crfD		0	L		A																				
cmpl	31		crfD		0	L		A						B								32						0
cmpli	10		crfD		0	L		A																				

Table A-4. Integer Logical Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
andx	31				S					A				B								28						Rc
andcx	31				S					A				B								60						Rc
andi.	28				S					A																		
andis.	29				S					A																		
cntlzwx	31				S					A				0 0 0 0 0								26						Rc
eqvx	31				S					A				B								284						Rc
extsbx	31				S					A				0 0 0 0 0								954						Rc
extshx	31				S					A				0 0 0 0 0								922						Rc
nandx	31				S					A				B								476						Rc
norx	31				S					A				B								124						Rc
orx	31				S					A				B								444						Rc
orcx	31				S					A				B								412						Rc
ori	24				S					A																		
oris	25				S					A																		
xorx	31				S					A				B								316						Rc
xori	26				S					A																		
xoris	27				S					A																		

Table A-5. Integer Rotate Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
rlwimix	20				S					A				SH							MB			ME				Rc
rlwinmx	21				S					A				SH							MB			ME				Rc
rlwnmx	23				S					A				SH							MB			ME				Rc

Table A-6. Integer Shift Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
slwx	31				S					A					B							24						Rc
srawx	31				S					A					B							792						Rc
srawix	31				S					A					SH							824						Rc
srwx	31				S					A					B							536						Rc

Table A-7. Floating-Point Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
faddx	63				D					A					B				00000				21					Rc
faddsx	59				D					A					B				00000				21					Rc
fdivx	63				D					A					B				00000				18					Rc
fdivsx	59				D					A					B				00000				18					Rc
fmulx	63				D					A				00000					C				25					Rc
fmulsx	59				D					A				00000					C				25					Rc
fresx	59				D				00000						B				00000				24					Rc
frsqrtx	63				D				00000						B				00000				26					Rc
fsubx	63				D					A					B				00000				20					Rc
fsubsx	59				D					A					B				00000				20					Rc
fselx	63				D					A					B				C				23					Rc

Table A-8. Floating-Point Multiply-Add Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fmaddx	63				D					A					B				C				29					Rc
fmaddsx	59				D					A					B				C				29					Rc
fmsubx	63				D					A					B				C				28					Rc
fmsubsx	59				D					A					B				C				28					Rc
fnmaddx	63				D					A					B				C				31					Rc
fnmaddsx	59				D					A					B				C				31					Rc
fnmsubx	63				D					A					B				C				30					Rc
fnmsubsx	59				D					A					B				C				30					Rc

Table A-9. Floating-Point Rounding and Conversion Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fctiw_x	63				D				0	0	0	0			B							14						Rc
fctiwz_x	63				D				0	0	0	0			B							15						Rc
frsp_x	63				D				0	0	0	0			B							12						Rc

Table A-10. Floating-Point Compare Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fcmpo	63				crfD		0	0			A				B							32						0
fcmpu	63				crfD		0	0			A				B							0						0

Table A-11. Floating-Point Status and Control Register Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
mcrfs	63				crfD		0	0			crfS		0	0		0	0	0	0			64						0
mffs_x	63				D				0	0	0	0			0	0	0	0				583						Rc
mtfsb0_x	63				crbD				0	0	0	0			0	0	0	0				70						Rc
mtfsb1_{xx}	63				crbD				0	0	0	0			0	0	0	0				38						Rc
mtfsf_x	63		0								FM		0		B							711						Rc
mtfsfix	63				crfD		0	0			0	0	0	0		IMM		0				134						Rc

Table A-12. Integer Load Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lbz	34				D					A											d							
lbzu	35				D					A											d							
lbzux	31				D					A					B							119						0
lbzx	31				D					A					B							87						0
lha	42				D					A											d							
lhau	43				D					A											d							
lhaux	31				D					A					B							375						0
lhax	31				D					A					B							343						0
lhz	40				D					A											d							
lhzu	41				D					A											d							
lhzux	31				D					A					B							311						0
lhzx	31				D					A					B							279						0
lwz	32				D					A											d							
lwzu	33				D					A											d							
lwzux	31				D					A					B							55						0
lwzx	31				D					A					B							23						0

Table A-13. Integer Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stb	38				S					A											d							
stbu	39				S					A											d							
stbux	31				S					A					B							247						0
stbx	31				S					A					B							215						0
sth	44				S					A											d							
sthu	45				S					A											d							
sthux	31				S					A					B							439						0
sthx	31				S					A					B							407						0
stw	36				S					A											d							
stwu	37				S					A											d							
stwux	31				S					A					B							183						0
stwx	31				S					A					B							151						0

Table A-14. Integer Load and Store with Byte Reverse Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lhbrx	31				D					A					B													0
lwbrx	31				D					A					B													0
sthbrx	31				S					A					B													0
stwbrx	31				S					A					B													0

Table A-15. Integer Load and Store Multiple Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lmw	46				D					A																		
stmw	47				S					A																		

Table A-16. Integer Load and Store String Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
lswi	31				D					A					NB													0
lswx	31				D					A					B													0
stswi	31				S					A					NB													0
stswx	31				S					A					B													0

Table A-17. Memory Synchronization Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
elcio	31				0	0	0	0	0			0	0	0	0	0												0
isync	19				0	0	0	0	0			0	0	0	0	0												0
lwarx	31				D					A					B													0
stwcx.	31				S					A					B													1
sync	31				0	0	0	0	0			0	0	0	0	0												0

Table A-18. Floating-Point Load Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
lfd	50				D					A											d								
lfd <u>u</u>	51				D					A											d								
lfd <u>ux</u>	31				D					A					B						631							0	
lfd <u>x</u>	31				D					A					B						599							0	
lfs	48				D					A											d								
lfs <u>u</u>	49				D					A											d								
lfs <u>ux</u>	31				D					A					B						567							0	
lfs <u>x</u>	31				D					A					B						535							0	

Table A-19. Floating-Point Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stfd	54				S					A											d							
stfdu	55				S					A											d							
stfdux	31				S					A					B						759						0	
stfdx	31				S					A					B						727						0	
stfiwx	31				S					A					B						983						0	
stfs	52				S					A											d							
stfsu	53				S					A											d							
stfsux	31				S					A					B						695						0	
stfsx	31				S					A					B						663						0	

Table A-20. Floating-Point Move Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
fabs<u>x</u>	63				D					0	0	0	0		B						264							Rc
fmr<u>x</u>	63				D					0	0	0	0		B						72							Rc
fnabs<u>x</u>	63				D					0	0	0	0		B						136							Rc
fneg<u>x</u>	63				D					0	0	0	0		B						40							Rc

Table A-21. Branch Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
bx	18	LI																												AA	LK
bcx	16	BO				BI				BD																		AA	LK		
bcctrx	19	BO				BI				0 0 0 0 0				528														LK			
bclrx	19	BO				BI				0 0 0 0 0				16														LK			

Table A-22. Condition Register Logical Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
crand	19	crbD				crbA				crbB				257														0	
crandc	19	crbD				crbA				crbB				129														0	
creqv	19	crbD				crbA				crbB				289														0	
crnand	19	crbD				crbA				crbB				225														0	
crnor	19	crbD				crbA				crbB				33														0	
cror	19	crbD				crbA				crbB				449														0	
crorc	19	crbD				crbA				crbB				417														0	
crxor	19	crbD				crbA				crbB				193														0	
mcrf	19	crfD			0 0		crfS			0 0		0 0 0 0 0			0 0 0 0 0 0 0 0 0 0														0

Table A-23. System Linkage Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
rfi ¹	19	0 0 0 0 0				0 0 0 0 0				0 0 0 0 0				50														0			
sc	17	0 0 0 0 0				0 0 0 0 0				0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																1	0				
Note:																															
1. Supervisor-level instruction																															

Table A-24. Trap Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
tw	31	TO				A				B				4														0	
twi	03	TO				A				SIMM																			

Table A-25. Processor Control Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
mcrxr	31	crfS			00		00000						00000						512						0						
mfcr	31	D			00000						00000						19						0								
mfmsr ¹	31	D			00000						00000						83						0								
mfspir ²	31	D			spr										339						0										
mftb	31	D			tpr										371						0										
mtcrf	31	S			0	CRM								0	144						0										
mtmsr ¹	31	S			00000						00000						146						0								
mtspir ²	31	D			spr										467						0										
Notes:																															
1. Supervisor-level instruction																															
2. Supervisor and user-level instruction																															

Table A-26. Cache Management Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
dcbf	31	0 0 0 0 0						A				B				86										0					
dcbi ¹	31	0 0 0 0 0						A				B				470										0					
dcbst	31	0 0 0 0 0						A				B				54										0					
dcbt	31	0 0 0 0 0						A				B				278										0					
dcbtst	31	0 0 0 0 0						A				B				246										0					
dcbz	31	0 0 0 0 0						A				B				1014										0					
icbi	31	0 0 0 0 0						A				B				982										0					
Note:																															
1. Supervisor-level instruction																															

Table A-27. Segment Register Manipulation Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
mfscr ¹	31				D			0		SR				0	0	0	0						595						0		
mfscrin ¹	31				D					0	0	0	0	0		B							659						0		
mtsr ¹	31				S			0		SR				0	0	0	0	0					210						0		
mtsrin ¹	31				S					0	0	0	0	0		B							242						0		
Note:																															
1. Supervisor-level instruction																															

Table A-28. Lookaside Buffer Management Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
tlbie ¹	31	0 0 0 0 0				0 0 0 0 0				B								306				0									
tlbsync ¹	31	0 0 0 0 0				0 0 0 0 0				0 0 0 0 0								566				0									
Note:																															
1. Supervisor-level instruction																															

Table A-29. External Control Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
eciwx	31	D				A				B				310												0		
ecowx	31	S				A				B				438												0		

Table A-30. Paired-Single Load and Store Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31			
psq_lx	4	D				A				B				w	i				6				0								
psq_stx	4	S				A				B				w	i				7				0								
psq_lux	4	D				A				B				w	i				38				0								
psq_stux	4	S				A				B				w	i				39				0								
psq_l	56	D				A				w	i				d																
psq_lu	57	D				A				w	i				d																
psq_st	60	S				A				w	i				d																
psq_stu	61	S				A				w	i				d																

Table A-31. Paired-Single Floating-Point Arithmetic Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ps_div	4				D					A					B					00000					18			Rc
ps_sub	4				D					A					B					00000					20			Rc
ps_add	4				D					A					B					00000					21			Rc
ps_sel	4				D					A					B					C					23			Rc
ps_res	4				D					00000					B					00000					24			Rc
ps_mul	4				D					A					00000					C					25			Rc
ps_rsqrt	4				D					00000					B					00000					26			Rc
ps_msub	4				D					A					B					C					28			Rc
ps_madd	4				D					A					B					C					29			Rc
ps_nmsub	4				D					A					B					C					30			Rc
ps_nmadd	4				D					A					B					C					31			Rc
ps_neg	4				D					00000					B										40			Rc
ps_mr	4				D					00000					B										72			Rc
ps_nabs	4				D					00000					B										136			Rc
ps_abs	4				D					00000					B										264			Rc

Table A-32. Miscellaneous Paired-Single Instructions

Name	0	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ps_sum0	4	D			A			B			C			10			Rc											
ps_sum1	4	D			A			B			C			11			Rc											
ps_muls0	4	D			A			00000			C			12			Rc											
ps_muls1	4	D			A			00000			C			13			Rc											
ps_madds0	4	D			A			B			C			14			Rc											
ps_madds1	4	D			A			B			C			15			Rc											
ps_cmpu0	4	crfD		00		A			B			0						0										
ps_cmpo0	4	crfD		00		A			B			32						0										
ps_cmpu1	4	crfD		00		A			B			64						0										
ps_cmpo1	4	crfD		00		A			B			96						0										
ps_merge00	4	D			A			B			528						Rc											
ps_merge01	4	D			A			B			560						Rc											
ps_merge10	4	D			A			B			592						Rc											
ps_merge11	4	D			A			B			624						Rc											
dcbz_l	4	00000			A			B			1014						0											

Glossary

ALU	Arithmetic logic unit
API	Abbreviated page index
APS	Adaptive power supply
BAT	Block address translation registers
BG	Bus Grant
BHT	Branch history table
BIU	Bus interface unit
BPU	Branch processing unit
BR	Bus Request
BTIC	Branch target instruction cache
CI	Cache Inhibit
CIA	Current instruction address
CIU	Core interface unit
CMOS	Complementary metal-oxide semiconductor
COP	Common on-chip processor
CR	Condition Register
CTR	Count Register
DABR	Data Address Breakpoint Register
DAR	Data Address Register
DBAT	Data BATs
DBG	Data bus grant
DEC	Decrementer Register
DMA	Direct Memory Access
DSI	Data storage interrupt
DSISR	Data Storage Interrupt Status Register
EA	Effective address
EAPI	Effective address page index
EAR	External Access Register

EDH	Extended data bus high
EDL	Extended data bus low
EDTI	Extended data transaction index
FIFO	First-in first-out
FPCC	Floating-point condition code
FPR	Floating-point register
FPSCR	Floating-Point Status and Control Register
FPU	Floating-point unit
GPR	General purpose register
GQR	Graphics Quantization Registers
HID n	Hardware Implementation-Dependent Register
HPM	High-performance mode signal name.
HRESET	Hard reset
IABR	Instruction Address Breakpoint Register
IBAT	instruction BATs
ICTC	Instruction Cache Throttling Control Register
INV	Invalid
IPL	Initial program load
IQ	Instruction queue
IU	Integer unit
JTAG	Joint Test Action Group
LR	Link Register
LRU	Least recently used
LSU	Load/store unit
MEI	Modified, exclusive, invalid
MERSI	Modified, exclusive, recent, shared, invalid
MESI	Modified, exclusive, shared, invalid
MIU	Memory interface unit
MMU	Memory management unit



MRU	Most recently used
MSR	Machine State Register
MUM	miss-under-miss
NIA	Next instruction address
OEA	Operating environment architecture
PCSR	Power Control and Status Register
PFH	Prefetch hint signal name
PI	Processor interface
PIR	Processor ID Register
PLL	Phase-locked loop
PLRU	Pseudo least-recently-used
POR	Power-on reset
PTE	Page table entry
PTEG	Page table entry groups
PVR	Processor Version Register
RA	Real address
RID	Resource ID
RISC	Reduced instruction set computer
RPN	Real page number
RWITM	Read-with-intent-to-modify
RWNITC	Read-with-no-intent-to-cache
SDA	Sampled Data Address Register
SIA	Sampled Instruction Address Register
SIMM	Signed immediate value
SPR	Special purpose register
SR _n	Segment register
SRESET	Soft reset
SRU	System register unit
TA	Transfer acknowledge signal name

TAP	Test access port
TAU	Thermal assist unit
TB	Time Base Register
TBU, TBL	Time Base Upper, Time Base Lower Registers
TBST	Transfer burst signal name
TCK	Test clock signal name
TDC	Thermal Diode Calibration Register
TDI	Test data input signal name
TDO	Test data output signal name
TLB	Translation lookaside buffer
TMS	Test mode select signal name
TRST	Test reset signal name
TRT	Transaction reorder table
TS	Transfer start signal name
TSIZ	Transaction size
TT	Transaction type
UDMAL	User Mode DMA Lower Register
UDMAU	User Mode DMA Upper Register
UIMM	Unsigned immediate value
UISA	User instruction set architecture
USDA	User Sampled Data Address Register
USIA	User Sampled Instruction Address Register
VAL	Valid
VEA	Virtual environment architecture
VID	Voltage ID
VPN	Virtual page number
VSID	Virtual segment ID
WIM	Write-through (W bit), caching-inhibited (I bit), memory coherency (M bit)
WIMG	Write-through (W bit), caching-inhibited (I bit), memory coherency (M bit), guarded memory (G bit)



WP	Write pipe
WPAR	Write Pipe Address Register
WPB	Write pipe buffer
XER	Fixed-Point Exception Register

